The Java™ Architecture for XML Binding (JAXB) 2.1

Final Release
December 11, 2006

Editors:
Kohsuke Kawaguchi,
Sekhar Vajjhala,
Joe Fialli
Comments to: spec-comments@jsr222.dev.java.net

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 USA

Specification: JSR-000222 Java(tm) Architecture for XML Binding ("Specification")
Version: 2.1
Status: Final Release
Release: 11 December 2006

LIMITED LICENSE GRANTS

1. _License for Evaluation Purposes_. Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Specification.

2. _License for the Distribution of Compliant Implementations_. Sun also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties). Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

3. _Pass-through Conditions_. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Sun's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification in question.

4. _Reciprocity Concerning Patent Licenses_.

a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would

be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

b. With respect to any patent claims owned by Sun and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against Sun that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

c. Also with respect to any patent claims owned by Sun and covered by the license granted under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Sun that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

5. _Definitions_. For the purposes of this Agreement: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Sun's source code or binary code materials nor, except with an appropriate and separate license from Sun, includes any of Sun's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.sun" or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Sun which corresponds to the Specification and that was available either (i) from Sun's 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

This Agreement will terminate immediately without notice from Sun if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED IN ANY WAY TO YOUR HAVING, IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

If you provide Sun with any comments or suggestions concerning the Specification ("Feedback"), you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Rev. April, 2006

Sun/Final/Full

CHAPTER 1

INTRODUCTION

XML is, essentially, a platform-independent means of structuring information. An XML document is a tree of *elements*. An element may have a set of *attributes*, in the form of key-value pairs, and may contain other elements, text, or a mixture thereof. An element may refer to other elements via *identifier* attributes or other types via *type* attributes, thereby allowing arbitrary graph structures to be represented.

An XML document need not follow any rules beyond the well-formedness criteria laid out in the XML 1.0 specification. To exchange documents in a meaningful way, however, requires that their structure and content be described and constrained so that the various parties involved will interpret them correctly and consistently. This can be accomplished through the use of a *schema*. A schema contains a set of rules that constrains the structure and content of a document's components, *i.e.*, its elements, attributes, and text. A schema also describes, at least informally and often implicitly, the intended conceptual meaning of a document's components. A schema is, in other words, a specification of the syntax and semantics of a (potentially infinite) set of XML documents. A document is said to be *valid* with respect to a schema if, and only if, it satisfies the constraints specified in the schema.

In what language is a schema defined? The XML specification itself describes a sublanguage for writing *document-type definitions*, or DTDs. As schemas go, however, DTDs are fairly weak. They support the definition of simple constraints on structure and content, but provide no real facility for expressing datatypes or complex structural relationships. They have also prompted the creation of more sophisticated schema languages such as XDR, SOX, RELAX, TREX, and, most significantly, the XML Schema language defined by the World Wide Web Consortium. The XML Schema language has gained widespread acceptance. It is the schema language of choice for many of the XML related specifications authored by industry standard working groups.

Therefore, the design center for this specification is W3C XML Schema language.

1.1 Data binding

Any nontrivial application of XML will, then, be based upon one or more schemas and will involve one or more programs that create, consume, and manipulate documents whose syntax and semantics are governed by those schemas. While it is certainly possible to write such programs using the low-level SAX parser API or the somewhat higher-level DOM parse-tree API, doing so is not easy. The resulting code is also difficult to maintain as the schemas evolve. While useful to some, many applications access and manipulate XML content within a document; its document structure is less relevant.

It would be much easier to write XML-enabled programs if we could simply map the components of an XML document to in-memory objects that represent, in an obvious and useful way, the document's intended meaning according to its schema. Of what classes should these objects be instances? In some cases there will be an obvious mapping from schema components to existing classes, especially for common types such as `String`, `Date`, `Vector`, and so forth. In general, however, classes specific to the schema being used will be required.

Classes specific to a schema may be derived or may already exist. In some application scenarios e.g. web services, a data model composed using user authored classes may already exist. A schema is derived from existing classes. In-memory objects are instances of existing classes. In other application scenarios, a data model is composed by authoring a schema. In such cases, rather than burden developers with having to write classes specific to schema, we can generate the classes directly from the schema. In all application scenarios, we create a Java object-level *binding* of the schema.

But even applications manipulating documents at conceptual object level, may desire to access or manipulate structural information in a document, e.g. element names. Therefore, the ability for an application to relate between the XML document representation and the Java object-level binding enables the use of XML operations over the XML document representation, e.g. Xpath.to bind XML content to an object model such as DOM is useful.

An *XML data-binding facility* therefore contains a *schema compiler and a schema generator.* A schema compiler can consume a schema and generate

schema derived classes specific to the schema. A schema generator consumes a set of existing classes and generates a schema.

A schema compiler binds components of a *source schema* to schema-derived Java *value classes*. Each value class provides access to the content of the corresponding schema component via a set of JavaBeans-style access (*i.e.*, `get` and `set`) methods. *Binding declarations* provides a capability to customize the binding from schema components to Java representation.

A schema generator binds existing classes to schema components. Each class containing Java Beans-style access (*i.e.*, `get` and `set`) methods is bound to a corresponding schema component. Program annotations provide a capability to customize the binding from existing classes to derived schema components. Access to content is through in-memory representation of existing classes.

Such a facility also provides a *binding framework*, a runtime API that, in conjunction with the derived or existing classes, supports three primary operations:

- The *unmarshalling* of an XML document into a tree of interrelated instances of both existing and schema-derived classes,
- The *marshalling* of such *content trees* back into XML documents, and
- The *binding,* maintained by a *binder,* between an XML document representation and *content tree*.

The unmarshalling process has the capability to check incoming XML documents for validity with respect to the schema.

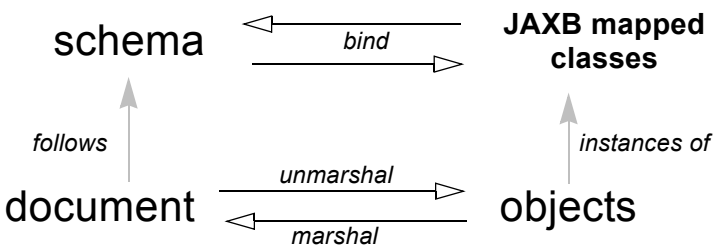


Figure 1.1 Binding XML to Java objects

To sum up: Schemas describe the structure and meaning of an XML document, in much the same way that a class describes an object in a program. To work with an XML document in a program we would like to map its components

directly to a set of objects that reflect the document's meaning according to its schema. We can achieve this by compiling the schema into a set of derived content classes or by compiling a set of existing classes into a schema and marshalling, unmarshalling and validating XML documents that follow the schema. Data binding thus allows XML-enabled programs to be written at the same conceptual level as the documents they manipulate, rather than at the more primitive level of parser events or parse trees.

Schema evolution in response to changing application requirements, is inevitable. A document therefore might not always necessarily follow the complete schema, rather a part of a versioned schema. Dealing with schema evolution requires both a versioning strategy as well as more flexible marshalling, unmarshalling and validation operations.

XML applications, such as workflow applications, consist of distributed, cooperative components interoperating using XML documents for data exchange. Each distributed component may need to access and update only a part of the XML document relevant to itself, while retaining the fidelity of the rest of the XML document. This is also more robust in the face of schema evolution, since the changes in schema may not impact the part of the document relevant to the application. The *binder* enables *partial binding* of the relevant parts of the XML document to a content tree and *marshalling* updates back to the original XML document.

1.2 Goals

The JAXB architecture is designed with the goals outlined here in mind.

1. **Full W3C XML Schema support**

   Support for XML Schema features not required to be supported in JAXB 1.0 has been added in this version.

2. **Binding existing Java classes to generated XML schema**

   This addresses application scenarios where design begins with Java classes rather than an XML schema. One such example is an application that exports itself as a web service that communicates using SOAP and XML as a transport mechanism. The marshalling of Java object graph is according to program annotations, either explicit or defaulted, on the existing Java classes.

3. **Meet data binding requirements for The Java API for XML Web Services(JAX-WS) 2.0**

   JAX-WS 2.0 will use the XML data binding facility defined by JAXB 2.0. Hence, JAXB 2.0 will meet all data binding requirements of JAX-WS 2.0.

4. **Ease of Development: Leverage J2SE 5.0 Language Extensions**

   For ease of development, J2SE 5.0 introduces additional language language extensions.The language extensions include generics (JSR 14), typesafe enums (JSR201), annotations (JSR 175). Use of the language extensions in binding of XML schema components to Java representation will result in a better and simpler binding, thus making the application development easier.

5. **Container Managed Environments**

   A container managed environment separates development from the deployment phases. This enables choice of generation of artifacts such as derived classes or derived schema at either development or deployment time.

   Any requirements related to the deployment of components using JAXB in a container managed environment will be addressed.

6. **Schema evolution**

   Schema evolution is a complex and diff 0 TDc0757 0 TD0.0082 Tc-0.0934 Tw[icsultarea i

Integration or relationship with the following Java technologies will be clarified.

a. Streaming API For XML (JSR 173) (StAX)

10. Relationship to XML related specifications

XML related specifications will be surveyed to determine their relationship to JAXB.

11. Portability of JAXB mapped classes

An architecture that provides for a fully portable JAXB 2.0 applications written to the J2SE platform will be defined. JAXB 2.0 is targeted for inclusion in a future version of J2SE. Application portability is a key requirement for inclusion in J2SE.

JAXB annotated classes must be source level and binary compatible with any other JAXB 2.0 binding framework implementation. As in JAXB 1.0, schema-derived interfaces/implementation classes are only required to be source code compatible with other JAXB implementations of the same version.

12. Preserving equivalence - Round tripping (Java to XML to Java)

Transforming a Java content tree to XML content and back to Java content again should result in an equivalent Java content tree before and after the transformation.

13. Preserving equivalence - Round tripping (XML to Java to XML)

While JAXB 1.0 specification did not require the preservation of the XML information set when round tripping from XML document to Java representation and back to XML document again, it did not forbid the preservation either. The same applies to this version of the specification.

14. Unmarshalling invalid XML content

Unmarshalling of invalid content was out of scope for JAXB 1.0. Simple binding rules and unmarshalling mechanisms that specify the handling of invalid content will be defined.

15. Ease of Use - Manipulation of XML documents in Java

Lower the barrier to entry to manipulating XML documents within Java programs. Programmers should be able to access and modify XML documents via a Java binding of the data, not via SAX or DOM. It should be possible for a developer who knows little about XML to compile a simple schema and immediately start making use of the

classes that are produced.

Rather than not supporting XML Schema extensibility concepts that can not be statically bound, such as unconstrained wildcard content, these concepts should be exposed directly as DOM or some other XML infoset preserving representation since there is no other satisfactory static Java binding representation for them.

16. **Customization**

    Applications sometimes require customization to meet their data binding requirements. Customization support will include:

    - XML to Java:

      A standard way to customize the binding of existing XML schema components to Java representation will be provided. JAXB 1.0 provided customization mechanisms for the subset of XML Schema components supported in JAXB 1.0. Customization support will be extended to additional XML Schema features to be supported in this version of the specification Section 1., “Full W3C XML Schema support,” on page 4.
    - Java to XML:

      A standard way to customize the binding of existing Java classes to XML schema will be added, Section 2., “Binding existing Java classes to generated XML schema,” on page 4.

17. **Schema derived classes should be natural**

    Insofar as possible, derived classes should observe standard Java API design guidelines and naming conventions. If new conventions are required then they should mesh well with existing conventions. A developer should not be astonished when trying to use a derived class.

18. **Schema derived classes should match conceptual level of source schema**

    It should be straightforward to examine any content-bearing component of the source schema and identify the corresponding Java language construct in the derived classes.

1.3 Non-Goals

- **Support for Java versions prior to J2SE 5.0**

  JAXB 2.0 relies on many of the Java language features added in J2SE 5.0. It is not a goal to support JAXB 2.0 on Java versions prior to J2SE 5.0.

- **Explicit support for specifying the binding of DTD to a Java representation**.

  While it was desired to explicitly support binding DTD to a Java representation, it became impractical to describe both XML Schema binding and DTD binding. The existence of several conversion tools that automate the conversion of a DTD to XML Schema allows DTD users to be able to take advantage of JAXB technology by converting their existing DTDs to XML Schema.

- **XML Schema Extensions**

  XML Schema specification allows the annotation of schemas and schema components with appinfo elements. JAXB 1.0 specifies the use of appinfo elements to customize the generated code. For this version of the specification, use of appinfo elements for customization of generated code continues to be in scope. However, use of appinfo element to introduce validation constraints beyond those already described in XML Schema 1.0 specification is out of scope.

- **Support for SOAP Encoding**

  SOAP Encoding is out of scope. Use of the SOAP encoding is essentially deprecated in the web services community, e.g. the WS-I Basic Profile[WSIBP] excludes SOAP encoding.

- **Support for validation on demand by schema derived classes**

  While working with a content tree corresponding to an XML document it is often necessary to validate the tree against the constraints in the source schema. JAXB 1.0 made it possible to do this at any time, without the user having to first marshal the tree into XML. However it is a non goal in JAXB 2.0, which leverages the JAXP 1.3 validation API.

- **Object graph traversal**

  Portable mechanisms to traverse a graph of JavaBean objects will not be addressed in JAXB 2.0.

- **Mapping any existing Java classes to any existing XML schema**

  The JAXB annotation mechanism is not sophisticated enough to enable mapping an arbitrary class to all XML schema concepts.

1.4 Conventions

Within normative prose in this specification, the words *should* and *must* are defined as follows:

- *should*
  Conforming implementations are permitted to but need not behave as described.
- *must*
  Conforming implementations are required to behave as described; otherwise they are in error.

Throughout the document, the XML namespace prefix `xs:` and `xsd:` refers to schema components in W3C XML Schema namespace as specified in [XSD Part 1] and [XSD Part 2]. The XML namespace prefix `xsi:` refers to the XML instance namespace defined in [XSD Part 1]. Additionally, the XML namespace prefix `jaxb`: refers to the JAXB namespace, `http://java.sun.com/xml/ns/jaxb`. The XML namespace prefix `ref:` refers to the namespace `http://ws-i.org/profiles/basic/1.1/xsd` as defined in [WSIBP] and [WSIAP].

All examples in the specification are for illustrative purposes to assist in understanding concepts and are non-normative. If an example conflicts with the normative prose, the normative prose always takes precedence over the example.

1.5 Expert Group Members

The following people have contributed to this specification.

Chavdar Baikov	SAP AG
David Bau	
Arnaud Blandin	
Stephen Brodsky	IBM
Russell Butek	IBM
Jongjin Choi	TMAX
Glen Daniels	Sonic Software
Blaise Doughan	Oracle
Christopher Fry	BEA Systems
Stanley Guan	Oracle
Mette Hedin	
Kohsuke Kawaguchi	Sun Microsystems, Inc.
Sravan Kumar	Pramati Technologies
Changshin Lee	Tmax Soft, Inc.
Anjana Manian	Oracle
Ed Merks	IBM
Steve Perry	Fidelity Information Services
Radu Preotiuc-Pietro	BEA
Srividya Rajagopalan	Nokia Corporation
Yann Raoul	
Bjarne Rasmussen	Novell, Inc.
Adinarayana Sakala	IONA Technologies PLC
Dennis M. Sosnoski	
Keith Visco	

Stefan Wachter	
Umit Yalcinalp	
Scott Ziegler	BEA Systems
Zulfi Umrani	Novell, Inc.

1.6 Acknowledgements

This document is a derivative work of concepts and an initial draft initially led by Mark Reinhold of Sun Microsystems. Our thanks to all who were involved in pioneering that initial effort. The feedback from the Java User community on the initial JAXB technology prototype greatly assisted in identifying requirements and directions.

The data binding experiences of the expert group members have been instrumental in identifying the proper blend of the countless data binding techniques that we have considered in the course of writing this specification. We thank them for their contributions and their review feedback.

Kohsuke Kawaguchi and Ryan Shoemaker have directly contributed content to the specification and wrote the companion javadoc. The following JAXB technology team members have been invaluable in keeping the specification effort on the right track: Tom Amiro, Leonid Arbouzov, Evgueni Astigueevitch, Jennifer Ball, Carla Carlson, Patrick Curran, Scott Fordin, Omar Fung, Peter Kacandes, Dmitry Khukhro, Tom Kincaid, K. Ari Krupnikov, Ramesh Mandava, Bhakti Mehta, Ed Mooney, Ilya Neverov, Oleg Oleinik, Brian Ogata, Vivek Pandey, Cecilia Peltier, Evgueni Rouban and Leslie Schwenk. The following people, all from Sun Microsystems, have provided valuable input to this effort: Roberto Chinnici, Chris Ferris, Mark Hapner, Eve Maler, Farrukh Najmi, Eduardo Pelegri-llopart, Bill Shannon and Rahul Sharma.

The JAXB TCK software team would like to acknowledge that the NIST XML Schema test suite [NIST] has greatly assisted the conformance testing of this specification.

1.7 Acknowledgements for JAXB 2.0

Version 2.0 of this specification was created under the Java Community Process as JSR-222. This specification is shaped by valuable input from expert group members, people with Sun, and Java User community feedback based on their experience with JAXB 1.0.

The data binding experience of the expert group has been very instrumental in identifying usage scenarios (including those from web services),design and evaluation of different databinding techniques. We thank them for their contributions and review feedback.

The following people, all from Sun Microsystems, have provided valuable input. The experiences of the reference implementation team, led by Kohsuke Kawaguchi, has been influential in identifying data binding solutions. Kohsuke Kawaguchi and Ryan Shoemaker have directly contributed content to the companion javadoc.Addtional feedback was provided by the following JAXB technology team members: Bhakti Mehta, Ed Mooney, Ryan Shoemaker, Karthikeyan Krishnamurthy, Tom Amiro, Leonid Arbouzov, Leonid Kuskov, Dmitry Fazunenko, Dmitry Lepekhin, Alexey Vishentsev, Omar Fung, and Anita Jindal. Valuable input was provided by the following people from Sun: Eduardo Pelegri-Llopart, Graham Hamilton, Mark Hapner, Bill Shannon.

The JAXB TCK software team would like to acknowledge that the NIST XML Schema test suite [NIST] has greatly assisted the conformance testing of this specification.

CHAPTER 2

REQUIREMENTS

This chapter specifies the scope and requirements for this version of the specification.

2.1 XML Schema to Java

2.1.1 W3C XML Schema support

The mapping of the following XML Schema components must be specified.

- type substitution (`@xsi:type`)
- element substitution group (<`xs:element @substitutionGroup`>)
- wildcard support(`xs:any` and `xs:anyAttribute`)
- identity constraints (`xs:key`, `xs:keyref` and `xs:unique`)
- redefinition of declaration (`xs:redefine`)
- NOTATION

For binding builtin XML Schema datatypes which do not map naturally to Java datatypes, Java datatypes specified by JAXP 1.3 (JSR 206) must be used.

2.1.2 Default Bindings

There must be a detailed, unambiguous description of the default mapping of schema components to Java representations in order to satisfy the portability goal.

2.1.3 Customized Binding Schema

A binding schema language and its formats must be specified. There must be a means to describe the binding without requiring modification to the original schema. Additionally, the same XML Schema language must be used for the two different mechanisms for expressing a binding declaration.

2.1.4 Overriding default binding behavior

Given the diverse styles that can be used to design a schema, it is daunting to identify a single ideal default binding solution. For situations where several equally good binding alternatives exist, the specification will describe the alternatives and select one to be the default (see Section 2.1.3, “Customized Binding Schema).

The binding schema must provide a means to specify an alternative binding for the scope of an entire schema. This mechanism ensures that if the default binding is not sufficient, it can easily be overridden in a portable manner.

2.1.5 JAX-WS 2.0

2.1.5.1 Backward Compatibility

Mapping of XML Schema to schema derived Java classes as specified in versions of JAX-RPC 1.x either by default or by customization is out of scope.

2.1.5.2 Binding XML Schema to schema derived classes

A binding of XML Schema constructs to schema derived classes must be specified. JAXB 1.0 specified the binding of XML Schema constructs to schema derived interfaces. However, JAX-RPC 1.x specified the binding of XML Schema constructs to schema derived classes, not interfaces. To provide continued support for JAX-RPC 1.x programming model, a customization to enable the binding of XML Schema constructs to schema derived classes will be added.

2.1.5.3 Accessing MIME content stored as an attachment

The W3C XMLP MTOM/XOP document and WS-I AP 1.0[WSIAP] both provide mechanisms for embedding references to SOAP attachments in SOAP messages. It is desirable to bind these to suitable Java types (e.g. Image or DataHandler) rather than just provide URI refs.

2.1.5.4 Serializable

A customization must be specified to enable schema derived classes to implement the `java.io.Serializable` interface. This capability enables the schema derived instance to be passed as EJB method parameter and to any other API that requires Serializable instances.

2.1.5.5 Disabling Databinding

A customization to disable databinding must be specified. When databinding is disabled, an XML Schema component is bound to an XML fragment representation rather than a strongly typed datatype determined by mapping rules. Binding to XML fragment allows the use of alternative binding technologies for example to perform XML operations.

2.2 Java to XML Schema

2.2.1 Default Mapping

A default mapping Java constructs to XML Schema must be specified. The default mapping may be overridden by customizations described in Section 2.2.2, “Customized Mapping”.

2.2.2 Customized Mapping

A customization mechanism to override default mapping of Java constructs to XML Schema constructs must be specified. Since XML Schema provides a much richer feature set than Java language for defining data models, the scope of customizations will be restricted to enable mapping to commonly used XML Schema constructs.

Support for the following mechanism must be specified:

- customization mechanism using the JSR175 program annotation facility.

2.2.3 JAX-WS 2.0

2.2.3.1 WSDL <types>

The WSDL <types> is generated using Java constructs to XML Schema mapping. The latter should therefore define customizations that enable mapping to XML Schema constructs commonly used in web services, subject to the requirements in Section 2.2.2, “Customized Mapping” and Section 2.2.1, “Default Mapping”.

2.2.3.2 Backward Compatibility

Mapping of existing Java constructs to XML Schema constructs as specified in JAX-RPC versions 1.x, either by default or through customization, is out of scope.

2.3 Binding Framework

2.3.1 Disabling schema validation

The specification will provide an ability to disable schema validation for unmarshal and marshal operations.

There exist a significant number of scenarios that do not require validation and/ or can not afford the overhead of schema validation. An application must be allowed to disable schema validation checking during unmarshal and marshal operations. The goal of this requirement is to provide the same flexibility and functionality that a SAX or DOM parser allows for.

CHAPTER 3

ARCHITECTURE

3.1 Introduction

This chapter describes the architectural components, comprising the XML-databinding facility, that realize the goals outlined in Section 1.2, “Goals”. The scope of this version of specification covers many additional goals beyond those in JAXB 1.0. As a result, JAXB 1.0 architecture has been revised significantly.

3.2 Overview

The XML data-binding facility consists of the following architectural components:

- ***schema compiler***: A schema compiler binds a *source schema* to a set of *schema derived program elements*. The binding is described by an XML-based language, ***binding language.***
- ***schema generator***: A schema generator maps a set of existing program elements to a *derived schema*. The mapping is described by ***program annotations.***
- ***binding runtime framework*** that provides two primary operations for accessing, manipulating and validating XML content using either schema derived or existing program elements:
  - *Unmarshalling* is the process of reading an XML document and constructing a tree of *content objects*. Each content object is an instance of either a schema derived or an existing program element mapped by the schema generator and corresponds to an instance in the XML document. Thus, the content tree reflects the document’s content.

Validation can optionally be enabled as part of the unmarshalling process. *Validation* is the process of verifying that an xml document meets all the constraints expressed in the schema.

- *Marshalling* is the inverse of unmarshalling, i.e., it is the process of traversing a content tree and writing an XML document that reflects the tree's content. Validation can optionally be enabled as part of the marshalling process.

As used in this specification, the term *schema* includes the W3C XML Schema as defined in the XML Schema 1.0 Recommendation[XSD Part 1][XSD Part 2]. Figure 3.1 illustrates relationships between concepts introduced in this section.

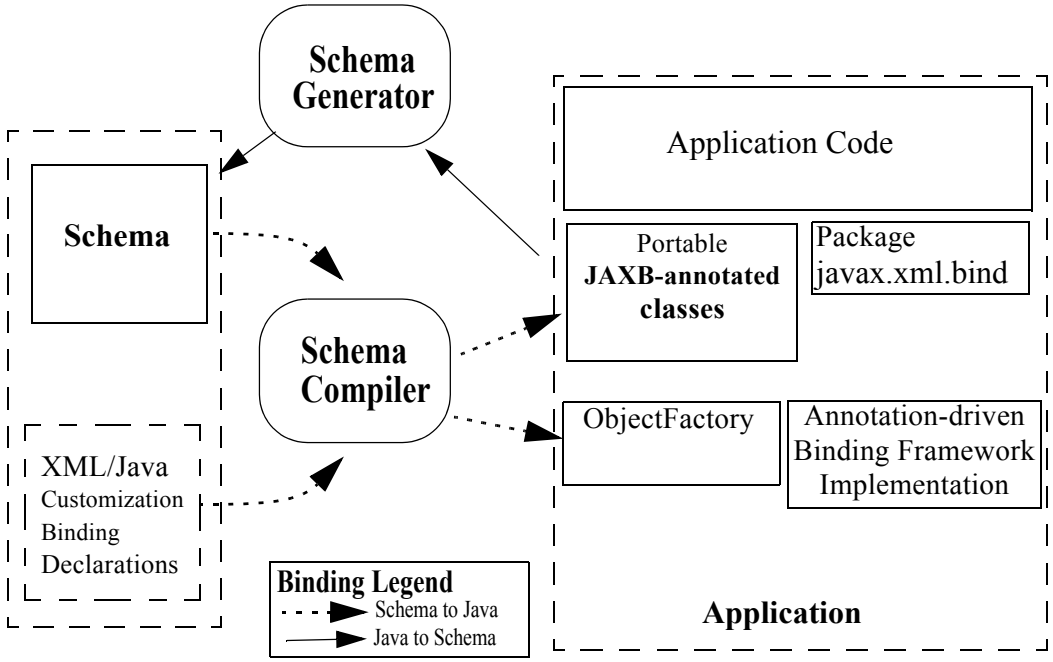
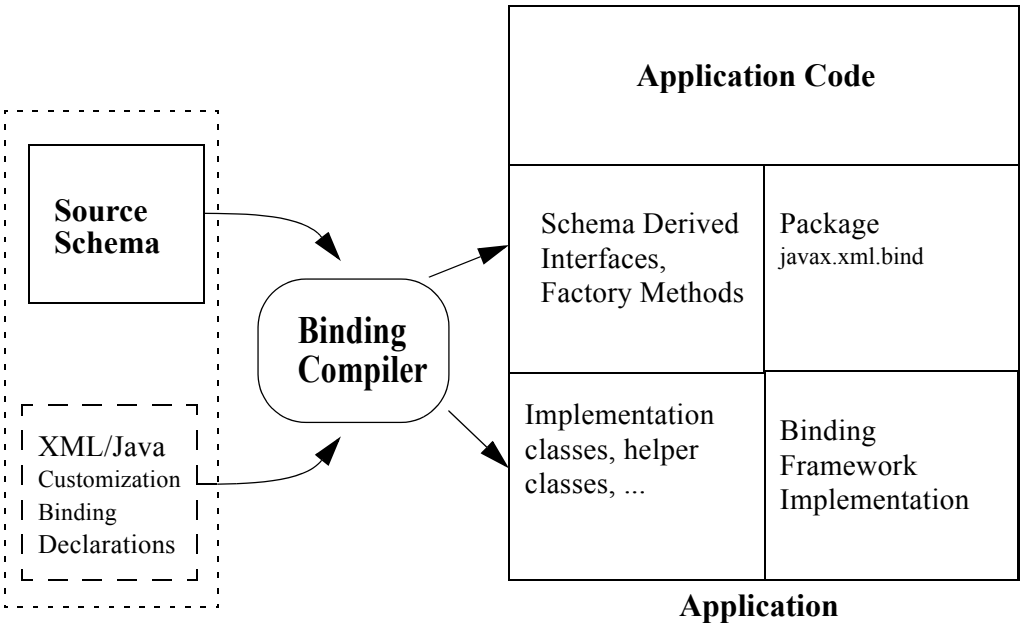


Figure 3.1 Non-Normative JAXB 2.0 Architecture diagram

JAXB-annotated classes are common to both binding schemes. They are either generated by a schema compiler or the result of a programmer adding JAXB annotations to existing Java classes. The universal unmarshal/marshal process is driven by the JAXB annotations on the portable JAXB-annotated classes.

Note that the binding declarations object in the above diagram is logical. Binding declarations can either be inlined within the schema or they can appear in an external binding file that is associated with the source schema.

Figure 3.2 JAXB 1.0 style binding of schema to interface/implementation classes.



Note that the application accesses only the schema-derived interfaces, factory methods and javax.xml.bind APIs directly. This convention is necessary to enable switching between JAXB implementations.

3.3 Java Representation

The content tree contains instances of *bound types*, types that bind and provide access to XML content. Each bound type corresponds to one or more schema components. As much as possible, for type safety and ease of use, a bound type that constrains the values to match the schema constraints of the schema components. The different bound types, which may be either schema derived or authored by a user, are described below.

Value Class A coarse grained schema component, such as a complex type definition, is bound to a Value class. The Java class hierarchy is used to preserve XML Schema's "derived by extension" type definition hierarchy. JAXB-annotated classes are portable and in comparison to schema derived interfaces/ implementation classes, result in a smaller number of classes.

Property A fine-grained schema component, such as an attribute declaration or an element declaration with a simple type, is bound directly to a *property* or a *field* within a value class.

A property is *realized* in a value class by a set of JavaBeans-style *access methods*. These methods include the usual `get` and `set` methods for retrieving and modifying a property's value; they also provide for the deletion and, if appropriate, the re-initialization of a property's value.

Properties are also used for references from one content instance to another. If an instance of a schema component *X* can occur within, or be referenced from, an instance of some other component *Y* then the content class derived from *Y* will define a property that can contain instances of *X*.

Binding a fine-grained schema component to a field is useful when a bound type does not follow the JavaBeans patterns. It makes it possible to map such types to a schema without the need to refactor them.

Interface JAXB 1.0 bound schema components (XML content) to schema derived content interfaces and implementation classes. The interface/ implementation classes tightly couple the schema derived implementation classes to the JAXB implementation runtime framework and are thus not portable. The binding of schema components to schema derived interfaces continues to be supported in JAXB 2.0.

Note – The mapping of existing Java interfaces to schema constructs is not supported. Since an existing class can implement multiple interfaces, there is no obvious mapping of existing interfaces to XML schema constructs.

Enum type J2SE 5.0 platform introduced linguistic support for type safe enumeration types. Enum type are used to represent values of schema types with enumeration values.

Collection type Collections are used to represent content models. Where possible, for type safety, parametric lists are used for homogeneous collections. For e.g. a repeating element in content model is bound to a parametric list.

DOM node In some cases, binding XML content to a DOM or DOM like representation rather than a collection of types is more natural to a programmer. One example is an open content model that allows elements whose types are not statically constrained by the schema.

Content tree can be created by unmarshalling of an XML document or by programmatic construction. Each bound type in the content tree is created as follows:

- schema derived implementation classes that implement schema derived interfaces can be created using factory methods generated by the schema compiler.
- schema derived value classes can be created using a constructor or a factory method generated by the schema compiler.
- existing types, authored by users, are required to provide a no arg constructor. The no arg constructor is used by an unmarshaller during unmarshalling to create an instance of the type.

3.3.1 Binding Declarations

A particular binding of a given source schema is defined by a set of *binding declarations*. Binding declarations are written in a *binding language*, which is itself an application of XML. A binding declaration can occur within the annotation `appinfo` of each XML Schema component. Alternatively, binding declarations can occur in an auxiliary file. Each binding declaration within the auxiliary file is associated to a schema component in the source schema. It was necessary to support binding declarations external to the source schema in order to allow for customization of an XML Schemas that one prefers not to modify. The schema compiler hence actually requires two inputs, a source schema and a set of binding declarations.

Binding declarations enable one to override default binding rules, thereby allowing for user customization of the schema-derived value class. Additionally, binding declarations allow for further refinements to be introduced into the binding to Java representation that could not be derived from the schema alone.

The binding declarations need not define every last detail of a binding. The schema compiler assumes *default binding declarations* for those components of the source schema that are not mentioned explicitly by binding declarations. Default declarations both reduce the verbosity of the customization and make it more robust to the evolution of the source schema. The defaulting rules are sufficiently powerful that in many cases a usable binding can be produced with no binding declarations at all. By defining a standardized format for the binding declarations, it is envisioned that tools will be built to greatly aid the process of customizing the binding from schema components to a Java representation.

3.3.2 Mapping Annotations

A mapping annotation defines the mapping of a program element to one or more schema components. A mapping annotation typically contains one or more *annotation members* to allow customized binding. An annotation member can be required or optional. A mapping annotation can be collocated with the program element in the source. The schema generator hence actually requires both inputs: a set of classes and a set of mapping annotations.

Defaults make it easy to use the mapping annotations. In the absence of a mapping annotation on a program element, the schema generator assumes, when required by a mapping rule, a *default mapping annotation.* This, together with an appropriate choice of default values for optional annotation members makes it possible to produce in many cases a usable mapping with minimal mapping annotations. Thus mapping annotations provide a powerful yet easy to use customization mechanism

3.4 Annotations

Many of the architectural components are driven by program annotations defined by this specification, *mapping annotations*.

Java to Schema Mapping Mapping annotations provide meta data that describe or customize the mapping of existing classes to a derived schema.

Portable Value Classes Mapping annotations provide information for unmarshalling and marshalling of an XML instance into a content tree representing the XML content without the need for a schema at run time. Thus schema derived code annotated with mapping annotations are portable i.e. they are capable of being marshalled and unmarshalled by a universal marshaller and unmarshaller written by a JAXB vendor implementation.

Adding application specific behavior and data Applications can choose to add either behavior or data to schema derived code. Section 6.13, “Modifying Schema-Derived Code” specifies how the mapping annotation, `@javax.annotation.Generated`, should be used by a developer to denote developer added/modified code from schema-derived code. This information can be utilized by tools to preserve application specific code across regenerations of schema derived code.

3.5 Binding Framework

The binding framework has been revised significantly since JAXB 1.0. Significant changes include:

- support for unmarshalling of invalid XML content.
- deprecation of on-demand validation.
- unmarshal/marshal time validation deferring to JAXP 1.3 validation.

3.5.1 Unmarshalling

3.5.1.1 Invalid XML Content

Rationale: Invalid XML content can arise for many reasons:

- When the cost of validation needs to be avoided.
- When the schema for the XML has evolved.
- When the XML is from a non-schema-aware processor.
- When the schema is not authoritative.

Support for invalid XML content required changes to JAXB 1.0 schema to java binding rules as well as the introduction of a flexible unmarshalling mode. These changes are described in Section 4.4.2, “Unmarshalling Modes”.

3.5.2 Validation

The constraints expressed in a schema fall into three general categories:

- A *type* constraint imposes requirements upon the values that may be provided by constraint facets in simple type definitions.
- A *local structural* constraint imposes requirements upon every instance of a given element type, e.g., that required attributes are given values and that a complex element’s content matches its content specification.
- A *global structural* constraint imposes requirements upon an entire document, e.g., that `ID` values are unique and that for every `IDREF`

attribute value there exists an element with the corresponding `ID` attribute value.

A *document* is valid if, and only if, all of the constraints expressed in its schema are satisfied. The manner in which constraints are enforced in a set of derived classes has a significant impact upon the usability of those classes. All constraints could, in principle, be checked only during unmarshalling. This approach would, however, yield classes that violate the *fail-fast* principle of API design: errors should, if feasible, be reported as soon as they are detected. In the context of schema-derived classes, this principle ensures that violations of schema constraints are signalled when they occur rather than later on when they may be more difficult to diagnose.

With this principle in mind we see that schema constraints can, in general, be enforced in three ways:

- *Static* enforcement leverages the type system of the Java programming language to ensure that a schema constraint is checked at application's compilation time. Type constraints are often good candidates for static enforcement. If an attribute is constrained by a schema to have a boolean value, e.g., then the access methods for that attribute's property can simply accept and return values of type `boolean`.
- *Simple dynamic* enforcement performs a trivial run-time check and throws an appropriate exception upon failure. Type constraints that do not easily map directly to Java classes or primitive types are best enforced in this way. If an attribute is constrained to have an integer value between zero and 100, e.g., then the corresponding property's access methods can accept and return `int` values and its mutation method can throw a run-time exception if its argument is out of range.
- *Complex dynamic* enforcement performs a potentially costly run-time check, usually involving more than one content object, and throwing an appropriate exception upon failure. Local structural constraints are usually enforced in this way: the structure of a complex element's content, e.g., can in general only be checked by examining the types of its children and ensuring that they match the schema's content model for that element. Global structural constraints must be enforced in this way: the uniqueness of `ID` values, e.g., can only be checked by examining the entire content tree.

It is straightforward to implement both static and simple dynamic checks so as to satisfy the fail-fast principle. Constraints that require complex dynamic checks could, in theory, also be implemented so as to fail as soon as possible.

The resulting classes would be rather clumsy to use, however, because it is often convenient to violate structural constraints on a temporary basis while constructing or manipulating a content tree.

Consider, e.g., a complex type definition whose content specification is very complex. Suppose that an instance of the corresponding value class is to be modified, and that the only way to achieve the desired result involves a sequence of changes during which the content specification would be violated. If the content instance were to check continuously that its content is valid, then the only way to modify the content would be to copy it, modify the copy, and then install the new copy in place of the old content. It would be much more convenient to be able to modify the content in place.

A similar analysis applies to most other sorts of structural constraints, and especially to global structural constraints. Schema-derived classes have the ability to enable or disable enable0.00 ethatleverifies type constraints. JAXB mapped classes can optionally be validated at unmarshal rshal time.

3.5.2.1 Validation Re architecture

The detection of complex schema constraint enableviolations has been redesigned to have a JAXB 2.0 implementation to delegate to the validation API in JAXP 1.3. JAXP 1.3 defines a standard enablevalidation API (javax.xml.validation package) for validating XML content t i nts within a schema. Furthermore, JAXP 1.3 has been incorporated into enableJ2SE 5.0 platform. Any JAXB 2.0 implementation that takes advantage of the validation API will result in a smaller footprint.

3.5.2.2 Unmarshal validation

When the unmarshalling process incorporates validation completes without any enablevalidation s, both the input document and the resulting content tree are guaranteed to be valid.

However, always requiring validation during unmarshalling proves to be too rigid and restrictive a requirement. Since existing XML parsers allow schema validation to be disabled, there exist a significant number of XML processing uses that disable schema validation to improve processing speed and/or to be able to process documents containing invalid or incomplete content. To enable the enableJAXB architecture to be used in these processing scenarios, the binding framework makes enablevalidation optional.

3.5.2.3 Marshal Validation

Validation may also be optionally performed at marshal time. This is new for JAXB 2.0. Validation of object graph while marshalling is useful in web services where the marshalled output must conform to schema constraints specified in a WSDL document. This could provide a valuable debugging aid for dealing with any interoperability problems

3.5.2.4 Handling Validation Failures

While it would be possible to notify a JAXB application that a validation error has occurred by throwing a `JAXBException` when the error is detected, this means of communicating a validation error results in only one failure at a time being handled. Potentially, the validation operation would have to be called as many times as there are validation errors. Both in terms of validation processing and for the application's benefit, it is better to detect as many errors and warnings as possible during a single validation pass. To allow for multiple validation errors to be processed in one pass, each validation error is mapped to a validation error event. A validation error event relates the validation error or warning encountered to the location of the text or object(s) involved with the error. The stream of potential validation error events can be communicated to the application either through a registered validation event handler at the time the validation error is encountered, or via a collection of validation failure events that the application can request after the operation has completed.

Unmarshalling and marshalling are the two operations that can result in multiple validation failures. The same mechanism is used to handle both failure scenarios. See Section 4.3, "General Validation Processing," on page 34 for further details.

3.6 An example

Throughout this specification we will refer and build upon the familiar schema from [XSD Part 0], which describes a purchase order, as a running example to illustrate various binding concepts as they are defined. Note that all schema name attributes with values in **this font** are bound by JAXB technology to either a Java interface or JavaBean-like property. Please note that the derived Java code in the example only approximates the default binding of the schema-to-Java representation.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
<xsd:element name="comment" type="xsd:string"/>
<xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
        <xsd:element name="shipTo"     type="USAddress"/>
        <xsd:element name="billTo"     type="USAddress"/>
        <xsd:element ref="comment"     minOccurs="0"/>
        <xsd:element name="items"      type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate"    type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="USAddress">
    <xsd:sequence>
        <xsd:element name="name"       type="xsd:string"/>
        <xsd:element name="street"     type="xsd:string"/>
        <xsd:element name="city"       type="xsd:string"/>
        <xsd:element name="state"      type="xsd:string"/>
        <xsd:element name="zip"        type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country"      type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>

<xsd:complexType name="Items">
    <xsd:sequence>
        <xsd:element name="item" minOccurs="1" maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="productName" type="xsd:string"/>
                    <xsd:element name="quantity">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:positiveInteger">
                                <xsd:maxExclusive value="100"/>
                            </xsd:restriction>
                        </xsd:simpleType>
                    </xsd:element>
                    <xsd:element name="USPrice"    type="xsd:decimal"/>
                    <xsd:element ref="comment"     minOccurs="0"/>
                    <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="partNum"      type="SKU" use="required"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>

<!-- Stock Keeping Unit, a code for identifying products -->
<xsd:simpleType name="SKU">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="\d{3}-[A-Z]{2}"/>
    </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Binding of purchase order schema to a Java representation[1]:

```
import javax.xml.datatype.XMLGregorianCalendar; import java.util.List;
public class PurchaseOrderType {
    USAddress    getShipTo(){...}          void    setShipTo(USAddress){...}
    USAddress    getBillTo(){...}          void    setBillTo(USAddress){...}
    /** Optional to set Comment property. */
    String       getComment(){...}         void    setComment(String){...}
    Items        getItems(){...}           void    setItems(Items){...}
    XMLGregorianCalendar getOrderDate() void setOrderDate(XMLGregorianCalendar)
};
public class USAddress {
    String       getName(){...}            void    setName(String){...}
    String       getStreet(){...}          void    setStreet(String){...}
    String       getCity(){...}            void    setCity(String){...}
    String       getState(){...}           void    setState(String){...}
    int          getZip(){...}             void    setZip(int){...}
    static final String COUNTRY="USA";[2]
};
public class Items {
    public class ItemType {
        String       getProductName(){...}    void    setProductName(String){...}
        /** Type constraint on Quantity setter value 0..99.[3]*/
        int          getQuantity(){...}       void    setQuantity(int){...}
        float        getUSPrice(){...}        void    setUSPrice(float){...}
        /** Optional to set Comment property. */
        String       getComment(){...}        void    setComment(String){...}
        XMLGregorianCalendar getShipDate();void setShipDate(XMLGregorianCalendar);
        /** Type constraint on PartNum setter value "\d{3}-[A-Z]{2}".[2]*/
        String       getPartNum(){...}        void    setPartNum(String){...}
    };

    /** Local structural constraint 1 or more instances of Items.ItemType.*/
    List<Items.ItemType> getItem(){...}
}
public class ObjectFactory {
    // type factories
    Object               newInstance(Class javaInterface){...}
    PurchaseOrderType    createPurchaseOrderType(){...}
    USAddress            createUSAddress(){...}
    Items                createItems(){...}
    Items.ItemType       createItemsItemType(){...}
    // element factories
    JAXBElement<PurchaseOrderType>createPurchaseOrder(PurchaseOrderType){...}
    JAXBElement<String>  createComment(String value){...}
}
```

1. In the interest of terseness, JAXB 2.0 program annotations have been ommitted.

2. Appropriate customization required to bind a fixed attribute to a constant value.

3. Type constraint checking only performed if customization enables it and implementation supports fail-fast checking

The purchase order schema does not describe any global structural constraints.

The coming chapters will identify how these XML Schema concepts were bound to a Java representation. Just as in [XSD Part 0], additions will be made to the schema example to illustrate the binding concepts being discussed.

CHAPTER 4

THE BINDING FRAMEWORK

The *binding framework* defines APIs to access unmarshalling, validation and marshalling operations for manipulating XML data and JAXB mapped objects. The framework is presented here in overview; its full specification is available in the javadoc for the package `javax.xml.bind`.

The binding framework resides in two main packages. The `javax.xml.bind` package defines abstract classes and interfaces that are used directly with content classes. The `javax.xml.bind`[1] package defines the `Unmarshaller, Marshaller` and `Binder` classes, which are auxiliary objects for providing their respective operations.

The `JAXBContext` class is the entry point for a Java application into the JAXB framework. A `JAXBContext` instance manages the binding relationship between XML element names to Java value class for a JAXB implementation to be used by the unmarshal, marshal and binder operations. The `javax.xml.bind.helper` package provides partial default implementations for some of the `javax.xml.bind` interfaces. Implementations of JAXB technology can extend these classes and implement the abstract methods. These APIs are not intended to be used directly by applications using the JAXB architecture. A third package, `javax.xml.bind.util`, contains utility classes that may be used directly by client applications.

The binding framework defines a hierarchy of exception and validation event classes for use when marshalling/unmarshalling errors occur, when constraints are violated, and when other types of errors are detected.

---

[1] JAXB 1.0 deprecated class, `javax.xml.bind.Validator`, is described in Section G.2.1, "Validator for JAXB 1.0 schema-derived classes".

4.1 Annotation-driven Binding Framework

The portability of JAXB annotated classes is achieved via an annotation-driven architecture. The program annotations, specified in Section 8, describe the mapping from the Java program elements to XML Schema components. This information is used by the binding framework to unmarshal and marshal to XML content into/from JAXB-annotated classes. All JAXB schema binding compilers must be able to generate portable schema-derived JAXB-annotated classes following the constraints described in Section 6, "Binding XML Schema to Java Representations". All binding runtime frameworks are required to be able to marshal and unmarshal portable JAXB-annotated classes generated by other JAXB 2.0 schema binding compiler.

It is not possible to require portability of the interface/implementation binding from JAXB 1.0. For backwards compatibility with existing implementations, that binding remains a tight coupling between the schema-derived implementation classes and the JAXB implementation's runtime framework. Users are required to regenerate the schema-derived implementation classes when changing JAXB implementations.

4.2 JAXBContext

The `JAXBContext` class provides the client's entry point to the JAXB API. It provides an abstraction for managing the XML/Java binding information necessary to implement the JAXB binding framework operations: unmarshal and marshal.

The following summarizes the `JAXBContext` class defined in package `javax.xml.bind`.

```
public abstract class JAXBContext {
    static final String JAXB_CONTEXT_FACTORY;
    static JAXBContext newInstance(String contextPath);
    static JAXBContext newInstance(String contextPath,
                                   ClassLoader contextPathCL);
    static JAXBContext newInstance(Class... classesToBeBound);
    abstract Unmarshaller createUnmarshaller();
    abstract Marshaller createMarshaller();
```

```
    abstract JAXBIntrospector createJAXBIntrospector();
    <T> Binder<T> createBinder(Class<T> domType);
    Binder<org.w3c.dom.Node> createBinder();
    void generateSchema(SchemaOutputResolver);
}
```

To avoid the overhead involved in creating a JAXBContext instance, a JAXB application is encouraged to reuse a JAXBContext instance. An implementation of abstract class JAXBContext is required to be thread-safe, thus, multiple threads in an application can share the same JAXBContext instance.

A client application configures a JAXBContext using the `JAXBContext.newInstance(String contextPath)` factory method.

```
JAXBContext jc =
       JAXBContext.newInstance( "com.acme.foo:com.acme.bar" );
```

The above example initializes a `JAXBContext` with the schema-derived Java packages `com.acme.foo` and `com.acme.bar`. A `jaxb.index` resource file, described in more detail in the javadoc, list the non-schema-derived classes, namely the java to schema binding, in a package to register with `JAXBContext`. Additionally, in each specified directory, if an optional resource file[2] containing package level mapping annotations exist, it is incorporated into the JAXBContext representation.

An alternative mechanism that could be more convenient when binding Java classes to Schema is to initialize JAXBContext by passing JAXB-annotated class objects.

```
JAXBContext jc =
    JAXBContext.newInstance( POElement.class );
```

The classes specified as parameters to `newInstance` and all classes that are directly/indirectly referenced statically from the specified classes are included into the returned `JAXBContext` instance. For each directory of all the classes imported into JAXBContext, if an optional resource file[2] containing package level mapping annotations exists, it is incorporated into the JAXBContext representation.

---

2. Section 7.4.1.1 "Package Annotations" in [JLS] recommends that file-system-based implementations have the annotated package declaration in a file called `package-info.java`.

For example, given the following Java classes:

```
@XmlRootElement[3] class Foo { Bar b;}
@XmlType class Bar { FooBar fb;}
@XmlType class FooBar { int x; }
```

The invocation of `JAXBContext.newInstance(Foo.class)` registers Foo and the statically referenced classes, `Bar` and `FooBar`.

Note that the jaxb.index resource file is not necessary when an application uses `JAXBContenxt.newInstances(Class...classesToBeBound)`.

For either scenario, the values of these parameters initialize the JAXBContext object so that it is capable of managing the JAXB mapped classes.

See the javadoc for `JAXBContext` for more details on using this class.

Design Note – JAXBContext class is designed to be immutable and thus thread-safe. Given the amount of dynamic processing that potentially could take place when creating a new instance of JAXBContxt, it is recommended that a JAXBContext instance be shared across threads and reused as much as possible to improve application performance.

4.3 General Validation Processing

Three identifiable forms of validation exist within the JAXB architecture include:

- **Unmarshal-time validation**

  This form of validation enables a client application to be notified of validation errors and warnings detected while unmarshalling XML data into a Java content tree and is completely orthogonal to the other types of validation. See javax.xml.bind.Unmarshaller javadoc for a description on how to enable this feature.

---

[3] Program annotations @XmlRootElement and @XmlType are specified in Section 8.0.

- **On-demand validation**

  This mode of validation is deprecated in JAXB 2.0. See Section G.2, "On-demand Validation" for the JAXB 1.0 description of this functionality.

- **Fail-fast validation**

  This form of validation enables a client application to receive immediate feedback about a modification to the Java content tree that violates a type constraint of a Java property. An unchecked exception is thrown if the value provided to a set method is invalid based on the constraint facets specified for the basetype of the property. This style of validation is optional in this version of the specification. Of the JAXB implementations that do support this type of validation, it is customization-time decision to enable or disable fail-fast validation when setting a property.

Unmarshal-time uses an event-driven mechanism to enable multiple validation errors and warnings to be processed during a single operation invocation. If the validation or unmarshal operation terminates with an exception upon encountering the first validation warning or error, subsequent validation errors and warnings would not be discovered until the first reported error is corrected. Thus, the validation event notification mechanism provides the application a more powerful means to evaluate validation warnings and errors as they occur and gives the application the ability to determine when a validation warning or error should abort the current operation (such as a value outside of the legal value space). Thus, an application could allow locally constrained validation problems to not terminate validation processing.

If the client application does not set an event handler on a `Unmarshaller` or `Marshaller` instance prior to invoking the `unmarshal`or `marshal` operations, then a default event handler will receive notification of any errors or fatal errors encountered and stop processing the XML data. In other words, the default event handler will fail on the first error that is encountered.

There are three ways to handle validation events encountered during the unmarshal and marshal operations:

- **Rely on the default validation event handler**
  The default handler will fail on the first error or fatal error encountered.
- **Implement and register a custom validation event handler**
  Client applications that require sophisticated event processing can

implement the `ValidationEventHandler` interface and register it with the Validator or Unmarshaller instance respectively.

- **Request an error/warning event list after the operation completes** By registering the `ValidationEventCollector` helper, a specialized event handler, with the `setEventHandler` method, the `ValidationEvent` objects created during the unmarshal and marshal operations are collected. The client application can then request the list after the operation completes.

Validation events are handled differently depending on how the client application is configured to process them as described previously. However, there are certain cases where a JAXB implementation may need to indicate that it is no longer able to reliably detect and report errors. In these cases, the JAXB implementation will set the severity of the `ValidationEvent` to `FATAL_ERROR` to indicate that the `unmarshal` or `validate` operation should be terminated. The default event handler and `ValidationEventCollector` helper class must terminate processing after being notified of a fatal error. Client applications that supply their own `ValidationEventHandler` should also terminate processing after being notified of a fatal error. If not, unexpected behavior may occur.

4.4 Unmarshalling

The `Unmarshaller` class governs the process of deserializing XML data into a Java content tree, capable of validating the XML data as it is unmarshalled. It provides the basic unmarshalling methods:

```
public interface Unmarshaller {
    ValidationEventHandler getEventHandler()
    void setEventHandler(ValidationEventHandler)

    java.lang.Object getProperty(java.lang.String name)
    void setProperty(java.lang.String name, java.lang.Object value)

    void setSchema(javax.xml.validation.Schema schema)
    javax.xml.validation.Schema getSchema()
```

```
        UnmarshallerHandler getUnmarshallerHandler()

        void setListener(Unmarshaller.Listener);
        Unmarshaller.Listener getListener();

        java.lang.Object unmarshal(java.io.File)
        java.lang.Object unmarshal(java.net.URL)
        java.lang.Object unmarshal(java.io.InputStream)
        java.lang.Object unmarshal(org.xml.sax.InputSource)
        java.lang.Object unmarshal(org.w3c.dom.Node)

        java.lang.Object unmarshal(javax.xml.transform.Source)
        java.lang.Object unmarshal(javax.xml.stream.XMLStreamReader)
        java.lang.Object unmarshal(javax.xml.stream.XMLEventReader)

        <T> JAXBElement<T> unmarshal(org.w3c.dom.Node,
                                     Class<T> declaredType)
        <T> JAXBElement<T> unmarshal(javax.xml.transform.Source,
                                     Class<T> declaredType)
        <T> JAXBElement<T> unmarshal(javax.xml.stream.XMLStreamReader,
                                     Class<T> declaredType)
        <T> JAXBElement<T> unmarshal(javax.xml.stream.XMLEventReader,
                                     Class<T> declaredType)
    }
```

The `JAXBContext` class contains a factory to create an `Unmarshaller` instance. The `JAXBContext` instance manages the XML/Java binding data that is used by unmarshalling. If the `JAXBContext` object that was used to create an `Unmarshaller` does not know how to unmarshal the XML content from a specified input source, then the `unmarshal` operation will abort immediately by throwing an `UnmarshalException`. There are six convenience methods for unmarshalling from various input sources.

An application can enable or disable unmarshal-time validation by enabling JAXP 1.3 validation via the `setSchema(javax.xml.validation.Schema)` method. The application has the option to customize validation error handling by overriding the default event handler using the `setEventHandler(ValidationEventHandler)`. The default event handler aborts the unmarshalling process when the first validation error event is encountered. Validation processing options are presented in more detail in Section 4.3, “General Validation Processing.”

An application has the ability to specify a SAX 2.0 parser to be used by the `unmarshal` operation using the `unmarshal(javax.xml.transform.Source)` method. Even though the JAXB provider's default parser is not required to be SAX2.0 compliant, all providers are required to allow an application to specify their own SAX2.0 parser. Some providers may require the application to specify the SAX2.0 parser at binding compile time. See the method javadoc `unmarshal(Source)` for more detail on how an application can specify its own SAX 2.0 parser.

The `getProperty/setProperty` methods introduce a mechanism to associate implementation specific property/value pairs to the unmarshalling process. At this time there are no standard JAXB properties specified for the unmarshalling process.

4.4.1 Unmarshal event callbacks

The `Unmarshaller` provides two styles of callback mechanisms that allow application specific processing during key points in the unmarshalling process. In 'class-defined' event callbacks, application specific code placed in JAXB mapped classes is triggered during unmarshalling. External listeners allow for centralized processing of unmarshal events in one callback method rather than by type event callbacks. The 'class defined' and external listener event callback methods are independent of each other, both can be called for one event. The invocation ordering when both listener callback methods exist is defined in `javax.xml.bind.Unmarshaller.Listener` javadoc.

Event callback methods should be written with following considerations. Each event callback invocation contributes to the overall unmarshal time. An event callback method throwing an exception terminates the current unmarshal process.

4.4.1.1 Class-defined

A JAXB mapped class can optionally implement the following unmarshal event callback methods.

```
private void beforeUnmarshal(Unmarshaller,
Object parent)
```

This method is called immediately after the object is created and before the unmarshalling of this object begins.The callback provides an opportunity to initialize JavaBean properties prior to unmarshalling.

Parameters:

unmarshaller - unmarshal context.

parent - points to the parent object to which this object will be set. Parent is null when this object is the root object.

```
private void afterUnmarshal(Unmarshaller,
Object parent)
```

This method is called after all the properties (except IDREF) are unmarshalled for this object, but before this object is set to the parent object.

Parameters:

unmarshaller - unmarshal context.

parent - points to the parent object to which this object will be set. Parent is null when this object is the root object.

These callback methods allow an object to perform additional processing at certain key point in the unmarshalling operation.

4.4.1.2 External Listener

The external listener callback mechanism enables the registration of a `Unmarshaller.Listener` instance with an `Unmarshaller.setListener(Unmarshaller.Listener)`. The external listener receives all callback events, allowing for more centralized processing than per class defined callback methods. The external listener receives events when unmarshalling to a JAXB element or to JAXB mapped class.

4.4.2 Unmarshalling Modes

There exist numerous use cases requiring the ability to unmarshal invalid XML content. A flexible unmarshalling mode is described in this version of the specification to enable predictable unmarshalling of invalid content. The previous unmarshalling mode implied by JAXB 1.0 specification is named structural unmarshalling. This unmarshalling mode was well defined for the unmarshalling of valid XML content and allowed an implementation to handle invalid XML content in anyway that it choose to.

Both of these modes have benefits and drawbacks based on an application's xml processing needs.

4.4.3 Structural Unmarshalling

Some of the XML Schema to Java bindings in JAXB 1.0 implied that an unmarshaller had to maintain a state machine, implying that the order of elements had to match up exactly as described by the schema or unmarshaller would work unpredictably. When this unmarshalling process detects a structural inconsistency that it is unable to recover from, it should abort the unmarshal process by throwing `UnmarshalException`.

For example, it was valid for a JAXB implementation to rigidly give up unmarshalling an invalid XML document once it came across an unexpected element/attribute or missed a required element or attribute. This mode appeals to users who prefer to be notified that an xml document is deviating from the schema.

XML Schema to Java binding for interfaces and implementation classes, Section 5.4.2, "Java Content Interface, can implement either structural unmarshalling or flexible unmarshalling.

4.4.4 Flexible Unmarshalling

To address the rigidness of structural unmarshalling, flexible unmarshalling mode is specified to enable greater predictability in unmarshalling invalid XML content. It unmarshals xml content by element name, rather than strictly on the position of the element within a content model. This allows this mode to handle the following cases:

- elements being out of order in a content model
- recovering from required elements/attributes missing from an xml document
- ignoring unexpected elements/attributes in an xml document

In order to enable this mode, the following JAXB 1.0 customized bindings that required state-driven unmarshalling have been removed from this specification.

- Binding a model group or model group definition to a Java class. Since there is no XML infoset information denoting these schema components, a model group can only be inferred by applying positional

schema constraints to a valid XML document, tracking position within a valid content model.

- Multiple occurrences of an element name in a content model can no longer be mapped to different JAXB properties. Instead the entire content model is bound to a general content model.

The removal of these bindings greatly assists the error recovery for structural unmarshalling mode.

Flexible unmarshalling appeals to those who need to be able to perform best match unmarshalling of invalid xml documents.

The flexible unmarshalling process is annotation driven. This process is specified in Appendix B, “Runtime Processing”. Flexible unmarshalling is required for JAXB annotated classes.

4.5 Marshalling

The `Marshaller` class is responsible for governing the process of serializing a Java content tree into XML data. It provides the basic marshalling methods:

```
interface Marshaller {
    static final string JAXB_ENCODING;
    static final string JAXB_FORMATTED_OUTPUT;
    static final string JAXB_SCHEMA_LOCATION;
    static final string JAXB_NO_NAMESPACE_SCHEMA_LOCATION;
    static final string JAXB_FRAGMENT;

    <PROTENTIALLY MORE PROPERTIES...>

    java.lang.Object getProperty(java.lang.String name)
    void setProperty(java.lang.String name, java.lang.Object value)

    void setEventHandler(ValidationEventHandler handler)
    ValidationEventHandler getEventHandler()

    void setSchema(javax.xml.validation.Schema schema)
    javax.xml.validation.Schema getSchema()

    void setListener(Unmarshaller.Listener)
    Unmarshaller.Listener getListener()
```

```
    void marshal(java.lang.Object e, java.io.Writer writer)
    void marshal(java.lang.Object e, java.io.OutputStream os)
    void marshal(java.lang.Object e, org.xml.sax.ContentHandler)
    void marshal(java.lang.Object e, javax.xml.transform.Result)
    void marshal(java.lang.Object e, org.w3c.dom.Node)
    void marshal(java.lang.Object e,
                 javax.xml.stream.XMLStreamWriter writer)

    org.w3c.dom.Node getNode(java.lang.Object contentTree)
}
```

The `JAXBContext` class contains a factory to create a `Marshaller` instance. Convenience method overloading of the `marshal()` method allow for marshalling a content tree to common Java output targets and to common XML output targets of a stream of SAX2 events or a DOM parse tree.

Although each of the marshal methods accepts a `java.lang.Object` as its first parameter, JAXB implementations are not required to be able to marshal any arbitrary `java.lang.Object`. If the first parameter is not a JAXB element, as determined by `JAXBIntrospector.isElement()` method, the marshal operation must throw a `MarshalException`. There exist two mechanisms to enable marshalling an instance that is not a JAXB element. One method is to wrap the instance as the value of a `javax.xml.bind.JAXBElement` instance, and pass the wrapper element as the first parameter to a `marshal` method. For java to schema binding, it is also possible to simply annotate the instance's class with the appropriate program annotation, `@XmlElementRoot`, specified in Section 8.

The marshalling process can optionally be configured to validate the content tree being marshalled. An application can enable or disable marshal-time validation by enabling JAXP 1.3 validation via the `setSchema(javax.xml.validation.Schema)` method. The application has the option to customize validation error handling by overriding the default event handler using the `setEventHandler(ValidationEventHandler)`. The default event handler aborts the marshalling process when the first validation error event is encountered. Validation processing options are presented in more detail in Section 4.3, “General Validation Processing.”

There is no requirement that the Java content tree be valid with respect to its original schema in order to marshal it back into XML data. If the marshalling process detects a structural inconsistency during its process that it is unable to

recover from, it should abort the marshal process by throwing `MarshalException`. The marshalling process of a JAXB-annotated class is annotation driven. This process is specified in Appendix B, “Runtime Processing”.

4.5.1 Marshal event callbacks

The Marshaller provides two styles of callback mechanisms that allow application specific processing during key points in the marshalling process. In class-defined event callbacks, application specific code placed in JAXB mapped classes is triggered during marshalling. External listeners allow for centralized processing of marshal events in one callback method rather than by type event callbacks. The invocation ordering when both listener callback methods exist is defined in `javax.xml.bind.Marshaller.Listener` javadoc.

Event callback methods should be written with following considerations. Each event callback invocation contributes to the overall marshal time. An event callback method throwing an exception terminates the current marshal process.

4.5.1.1 Class-defined

A JAXB mapped class can optionally implement the following marshal event callback methods.

`private void beforeMarshal(Marshaller)`

This method is called before the marshalling of this object starts.

Parameters:

marshaller - marshal context.

`private void afterMarshal(Marshaller)`

This method is called after the marshalling of this object (and all its descendants) has finished.

Parameters:

marshaller - marshal context.

These callback methods allow the customization of an JAXB mapped class to perform additional processing at certain key point in the marshalling operation. The 'class defined' and external listener event callback methods are independent of each other, both can be called for one event.

An event callback method throwing an exception terminates the current marshal process.

4.5.1.2 External Listener

The external listener callback mechanism enables the registration of a `Marshaller.Listener` instance with a `Marshaller.setListener(Marshaller.Listener)`. The external listener receives all callback events, allowing for more centralized processing than per class-defined callback methods.

4.5.2 Marshalling Properties

The following subsection highlights properties that can be used to control the marshalling process. These properties must be set prior to the start of a marshalling operation: the behavior is undefined if these attributes are altered in the middle of a marshalling operation. The following standard properties have been identified:

- `jaxb.encoding`: output character encoding. If the property is not specified, it defaults to "UTF-8".
- `jaxb.formatted.output`:
  `true` - human readable indented xml data
  `false` - unformatted xml data
  If the property is not specified, it defaults to `false`.
- `jaxb.schemaLocation`
  This property allows the client application to specify an `xsi:schemaLocation` attribute in the generated XML data.
- `jaxb.noNamespaceSchemaLocation`
  This property allows the client application to specify an `xsi:noNamespaceSchemaLocation` attribute in the generated XML data.
- `jaxb.fragment`
  Its value must be a java.lang.Boolean. This property determines whether or not document level events will be generated by the Marshaller. If this property is not defined, it defaults to `false`.

4.6 JAXBIntrospector

This class provides access to key XML mapping information of a JAXB mapped instance.

```
public abstract class JAXBIntrospector {
    public boolean isElement(Object jaxbObj);
    public QName getElementName(Object jaxbElement);
    public static Object getValue(Object jaxbElement);
}
```

The JAXB 2.0 architecture has two uniquely different ways to represent an XML element.The XML Schema to Java binding for an XML element declaration is described in Section 5.6, “Java Element Representation”. The Java to XML Schema binding for an XML element declaration is described in Section 8.8.2, “@XmlRootElement.“

Use JAXBInstrospector.isElement(Object) method to determine if an instance of a JAXB mapped class represents an XML element. One can get the xml element tag name associated with a JAXB element using `JAXBIntrospector.getElementName` method.One can an xml element’s value using getValue method. The getValue method normalizes access of JAXB element, hiding whether the JAXB element is an instance of javax.xml.bind.JAXBElement or if it is an JAXB element via an @XmlRootElement class annotation.

4.7 Validation Handling

Methods defined in the binding framework can cause validation events to be delivered to the client application’s `ValidationEventHandler`. `Setter` methods generated in schema-derived classes are capable of throwing `TypeConstraintExceptions`, all of which are defined in the binding framework.

The following list describes the primary event and constraint-exception classes:

- An instance of a `TypeConstraintException` subclass is thrown when a violation of a dynamically-checked type constraint is detected. Such exceptions will be thrown by property-set methods, for which it

would be inconvenient to have to handle checked exceptions; type-constraint exceptions are therefore unchecked, *i.e*, this class extends `java.lang.RuntimeException`. The constraint check is always performed prior to the property-set method updating the value of the property, thus if the exception is thrown, the property is guaranteed to retain the value it had prior to the invocation of the property-set method with an invalid value. This functionality is optional to implement in this version of the specification. Additionally, a customization mechanism is provided to control enabling and disabling this feature.

- An instance of a `ValidationEvent` is delivered whenever a violation is detected during optionally enabled unmarshal/marshal validation. Additionally, `ValidationEvents` can be discovered during marshalling such as ID/IDREF violations and print conversion failures. These violations may indicate local and global structural constraint violations, type conversion violations, type constraint violations, etc.
- Since the unmarshal operation involves reading an input document, lexical well-formedness errors may be detected or an I/O error may occur. In these cases, an `UnmarshalException` will be thrown to indicate that the JAXB provider is unable to continue the unmarshal operation.
- During the marshal operation, the JAXB provider may encounter errors in the Java content tree that prevent it from being able to complete. In these cases, a `MarshalException` will be thrown to indicate that the marshal operation can not be completed.

4.8 DOM and Java representation Binding

The Binder class is responsible for maintaining the relationship between a infoset preserving view of an XML document with a possibly partial binding of the XML document to a JAXB representation. Modifications can be made to either the infoset preserving view or the JAXB representation of the document while the other view remains unmodified. The binder is able to synchronize the changes made in the modified view back into the read-only view. When synchronizing changes to JAXB view back to related xml infoset preserving view, every effort is made to preserve XML concepts that are not bound to

JAXB objects, such as XML infoset comments, processing instructions, namespace prefix mappings, etc.

4.8.1 Use Cases

- Read-only partial binding.

  Application only needs to manipulate a small part of a rather large XML document. It suffices to only map the small of the large document to the JAXB Java representation.

- Updateable partial binding

  The application receives an XML document that follows a later version of the schema than the application is aware of. The parts of the schema that the application needs to read and/or modify have not changed. Thus, the document can be read into an infoset preserving representation, such as DOM, only bind the part of the document that it does still have the correct schema for into the JAXB Java representation of the fragment of the document using Binder.unmarshal from the DOM to the JAXB view. Modify the partial Java representation of the document and then synchronize the modified parts of the Java representation back to the DOM view using `Binder.updateXML` method.

- XPATH navigation

  Given that binder maintains a relationship between XML infoset view of document and JAXB representation, one can use JAXP 1.3 XPATH on the XML infoset view and use the binder's associative mapping to get from the infoset node to JAXB representation.

4.8.2 javax.xml.bind.Binder

The class `javax.xml.bind.Binder` associates an infoset preserving representation of the entire XML document with a potentially partial binding to a Java representation. The binder provides operations to synchronize between the two views that it is binding.

```
public abstract class Binder<XmlNode> {
    // Create two views of XML content, infoset view and JAXB view
    public abstract Object unmarshal(XmlNode)
    <T> JAXBElement<T> unmarshal(org.w3c.dom.Node,
                                 Class<T> declaredType)
    public abstract void marshal(Object jaxbObjcet, XmlNode)

    //Navigation between xml infoset view and JAXB view.
    public abstract XmlNode getXMLNode(Object jaxbObject );
    public abstract Object getJAXBNode(XmlNode);

    // Synchronization methods
    public abstract XmlNode updateXML(Object jaxbObject )
    public abstract XmlNode updateXML(Object jaxbObject, XmlNode)
        throws JAXBException;
    public abstract Object updateJAXB( XmlNode)
        throws JAXBException;

    // Enable optional validation
    public abstract void setSchema(Schema);
    public abstract Schema getSchema();
    public abstract void setEventHandler( ValidationEventHandler)
        throws JAXBException;
    public abstract ValidationEventHandler getEventHandler()
        throws JAXBException;

    // Marshal/Unmarshal properties
    public abstract void setProperty(String name, Object value)
        throws PropertyException;
    public abstract Object getProperty(String name)
        throws PropertyException;
}
```

CHAPTER 5

JAVA REPRESENTATION OF XML CONTENT

This section defines the basic binding representation of package, value class, element classes, properties and enum type within the Java programming language. Each section briefly states the XML Schema components that could be bound to the Java representation. A more rigorous and thorough description of possible bindings and default bindings occurs in Chapter 6, "Binding XML Schema to Java Representations" and in Chapter 7, "Customizing XML Schema to Java Representation Binding."

5.1 Mapping between XML Names and Java Identifiers

XML schema languages use *XML names*, *i.e.*, strings that match the Name production defined in XML 1.0 (Second Edition) to label schema components. This set of strings is much larger than the set of valid Java class, method, and constant identifiers. Appendix D, "Binding XML Names to Java Identifiers," specifies an algorithm for mapping XML names to Java identifiers in a way that adheres to standard Java API design guidelines, generates identifiers that retain obvious connections to the corresponding schema, and results in as few collisions as possible. It is necessary to rigorously define a standard way to perform this mapping so all implementations of this specification perform the mapping in the same compatible manner.

5.2 Java Package

Just as the target XML namespace provides a naming context for the named type definitions, named model groups, global element declarations and global attribute declarations for a schema vocabulary, the Java package provides a naming context for Java interfaces and classes. Therefore, it is natural to map the target namespace of a schema to be the package that contains the Java value class representing the structural content model of the document.

A package consists of:

- A *name*, which is either derived directly from the XML namespace URI as specified in Section D.5, "Generating a Java package name" or specified by a binding customization of the XML namespace URI as described in Section 7.6.1.1, "package."
- A set of Java value class representing the content models declared within the schema.
- A set of Java element classes representing element declarations occurring within the schema. Section 5.6, "Java Element Representation" describes this binding in more detail.
- A public class `ObjectFactory` contains:
  - An instance factory method signature for each Java content within the package.

    Given Java value class named *Foo*, here is the derived factory method:

    ```
    public Foo createFoo();
    ```

  - An element instance factory method signature for each bound element declaration.

    ```
    public JAXBElement<T> createFoo(T elementValue);
    ```

  - Dynamic instance factory allocator method signature:

    ```
    public Object newInstance(Class javaContentInterface);
    ```

  - Property setter/getter
    Provide the ability to associate implementation specific property/value pairs with the instance creation process.

```
        java.lang.Object getProperty(String name);
        void setProperty(String name, Object value);
```

- A set of enum types.
- Package javadoc.

Example:

Purchase Order Schema fragment with `targetNamespace`:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
              xmlns:po="http://www.example.com/PO1"
              targetNamespace="http://www.example.com/PO1">
   <xs:element name="purchaseOrder" type="po:PurchaseOrderType"/>
   <xs:element name="comment"       type="xs:string"/>
   <xs:complexType name="PurchaseOrderType"/>
   ...
</xs:schema>
```

Default derived Java code:

```
import javax.xml.bind.JAXBElement;
package com.example.PO1;
public class PurchaseOrderType { .... };
public Comment { String getValue(){...} void setValue(String){...} }
...
public class ObjectFactory {
    PurchaseOrderType createPurchaseOrderType();
    JAXBElement<PurchaseOrderType>
createPurchaseOrder(PurchaseOrderType elementValue);
    Comment createComment(String value);
    ...
}
```

5.3 Enum Type

A simple type definition whose value space is constrained by enumeration facets can be bound to a Java enum type. Enum type was introduced in J2SE 5.0 and is described in Section 8.9 of [JLS]. Enum type is a significant enhancement over the typesafe enum design pattern that it was designed to replace. If an application wishes to refer to the values of a class by descriptive constants and

manipulate those constants in a type safe manner, it should consider binding the XML component containing enumeration facets to an enum type.

An enum type consists of:

- A *name*, which is either computed directly from an XML name or specified by a binding customization for the schema component.
- A *package name*, which is either computed from the target namespace of the schema component or specified within a binding declaration as a customization of the target namespace or a specified package name for components that are scoped to no target namespace.
- Outer Class Names is "**.**" separated list of outer class names.

  By default, if the XML component containing a typesafe enum class to be generated is scoped within a complex type as opposed to a global scope, the typesafe enum class should occur as a nested class within the Java value class representing the complex type scope.
  Absolute class name is PackageName.[OuterClassNames.]Name.
  Note: Outer Class Name is null if class is a top-level class.

  The schema customization <jaxb:globalBindings localScoping="toplevel"/>, specified in Section 7.5.1, disables the generation of schema-derived nested classes and can be used to override the default binding of a nested schema component binding to nested Java class.
- A set of *enum constants.*
- Class javadoc is a combination of a documentation annotation from the schema component and/or javadoc specified by customization.

An *enum constant* consists of:

- A *name*, which is either computed from the enumeration facet value or specified by customization.
- A *value* for the constant. Optimally, the *name* is the same as the *value*. This optimization is not possible when the enumeration facet value is not a valid Java identifier.
- A *datatype* for the constant's value.
- *Javadoc for the constant field* is a combination of a documentation annotation for an enumeration value facet and/or javadoc specified by customization.

5.4 Content Representation

A complex type definition is bound to either a Java value class or a content interface, depending on the value of the global binding customization **[jaxb:globalBinding]** `@generateValueClass`, specified in Section 7.5.1, "Usage". Value classes are generated by default. The attributes and children element content of a complex type definition are represented as properties of the Java content representation. Property representations are introduced in Section 5.5, "Properties," on page 54.

5.4.1 Value Class

A value class consists of:

- A *name*, which is either computed directly from an XML name or specified by a binding customization for the schema component.
- A *package name*, which is either computed from the target namespace of the schema component or specified by a binding customization of the target namespace or a specified package name for components that are scoped to no target namespace.
- The *outer class name* context, a dot-separated list of Java class names.

  By default, if the XML schema component for which a Java value class is to be generated is scoped within a complex type as opposed to globally, the complex class should occur as a nested class within the Java value class representing the complex type scope. The schema customization <jaxb:globalBindings localScoping="toplevel"/>, specified in Section 7.5.1, disables the generation of schema-derived nested classes and all classes are generated as toplevel classes.

  The absolute class name is PackageName.[OuterClassNames.]Name. Note: The OuterClassNames is null if the class is a top-level class.
- A base class that this class extends. See Section 6.3, "Complex Type Definition," on page 90 for further details.
- A set of Java properties providing access and modification to the complex type definition's attributes and content model represented by the value class.
- Class-level javadoc is a combination of a documentation annotation from the schema component and/or javadoc specified within customization.

- Creation
  - A value class supports creation via a public constructor, either an explicit one or the default no-arg constructor.
  - A factory method in the package's `ObjectFactory` class (introduced in Section 5.2, "Java Package"). The factory method returns the type of the Java value class. The name of the factory method is generated by concatenating the following components:
    - The string constant `create`.
    - If the Java value class is nested within another value class, then the concatenation of all outer Java class names.
    - The *name* of the Java value class.

    For example, a Java value class named `Foo` that is nested within Java value class `Bar` would have the following factory method signature generated in the containing Java package's `ObjectFactory` class:

    ```
    Bar.Foo createBarFoo() {...}
    ```

5.4.2 Java Content Interface

JAXB 1.0 bound a complex type definition to a content interface. This binding is similar to the value class binding with the following differences.

- A content interface is a public interface while a value class is a public class.
- A content interface can only be created with an ObjectFactory method whereas a value class can be created using a public constructor. The factory method signature is the same for both value class and content interface binding to ease switching between the two binding styles.
- A content interface contains the method signatures for the set of properties it contains, while a value class contains method implementations.

5.5 Properties

The schema compiler binds local schema components to *properties* within a Java value class.

A property is defined by:

- A *name*, which is either computed from the XML name or specified by a binding customization for the schema component.
- A *base type*, which may be a Java primitive type (*e.g.*, `int`) or a reference type.
- An optional *predicate*, which is a mechanism that tests values of the base type for validity and throws a `TypeConstraintException` if a type constraint expressed in the source schema is violated. [1]
- An optional *collection type*, which is used for properties whose values may be composed of more than one value.
- A *default value*. Schema component has a schema specified default value which is used when property's value is not set and not nil.
- Is *nillable*. A property is nillable when it represents a nillable element declaration.

A property is *realized* by a set of *access methods*. Several property models are identified in the following subsections; each adds additional functionally to the basic set of access methods.

A property's access methods are named in the standard JavaBeans style: the name-mapping algorithm is applied to the property name and then each method name is constructed by prefixing the appropriate verb (`get`, `set`, etc.).

A property is said to have a *set value* if that value was assigned to it during unmarshalling[2] or by invoking its mutation method. The *value* of a property is its *set value*, if defined; otherwise, it is the property's schema specified *default value*, if any; otherwise, it is the default initial value for the property's base type as it would be assigned for an uninitialized field within a Java class[3]. Figure 5.1 illustrates the states of a JAXB property and the invocations that result in state changes.

1. Note that it is optional for a JAXB implementation to support type constraint checks when setting a property in this version of the specification.

2. An unmarshalling implementation should distinguish between a value from an XML instance document and a schema specified defaulted value when possible. A property should only be considered to have a *set value* when there exists a corresponding value in the XML content being unmarshalled. Unfortunately, unmarshalling implementation paths do exist that can not identify schema specified default values, this situation is considered a one-time transformation for the property and the defaulted value will be treated as a *set value*.

3. Namely, a `boolean` field type defaults to `false`, `integer` field type defaults to `0`, object reference field type defaults to `null`, floating point field type defaults to `+0.0f`.

5.5.1 Simple Property

A non-collection property `prop` with a base type *Type* is realized by the two methods

```
public Type getId ();
public void setId (Type value);
```

where *Id* is a metavariable that represents the Java method identifier computed by applying the name mapping algorithm described in Section D.2, "The Name to Identifier Mapping Algorithm" to prop. There is one exception to this general rule in order to support the boolean property described in [BEANS]. When *Type* is boolean, the `getId` method specified above is replaced by the method signature, *boolean* `isId`().

- The `get` or `is` method returns the property's value as specified in the previous subsection. If *null* is returned, the property is considered to be absent from the XML content that it represents.
- The `set` method defines the property's *set value* to be the argument `value`. If the argument value is `null`, the property's *set value* is discarded. Prior to setting the property's value when TypeConstraint validation is enabled[4], a non-`null` value is validated by applying the property's predicate. If `TypeConstraintException` is thrown, the property retains the value it had prior to the `set` method invocation.

When the base type for a property is a primitive non-reference type and the property's value is optional, the corresponding Java wrapper class can be used as the base type to enable discarding the property's set value by invoking the set method with a null parameter. Section 5.5.4, "`isSet` Property Modifier," on page 61 describes an alternative to using a wrapper class for this purpose. The **`[jaxb:globalBinding]`** customization `@optionalProperty` controls the binding of an optional primitive property as described in Section 7.5.1, "Usage.

[4] Note that it is optional for a JAXB implementation to support type constraint checks when setting a property in this version of the specification.

Example

In the purchase order schema, the `partNum` attribute of the `item` element definition is declared:

```
<xs:attribute name="partNum" type="SKU" use="required"/>
```

This element declaration is bound to a simple property with the base type `java.lang.String`:

```
public String getPartNum();
public void setPartNum(String x);
```

The `setPartNum` method could apply a predicate to its argument to ensure that the new value is legal, *i.e.*, that it is a string value that complies with the constraints for the simple type definition, SKU, and that derives by restriction from `xs:string` and restricts the string value to match the regular expression pattern `"\d{3}-[A-Z]{2}"`.

It is legal to pass `null` to the `setPartNum` method even though the `partNum` attribute declaration's attribute `use` is specified as required. The determination if `partNum` content actually has a value is a local structural constraint rather than a type constraint, so it is checked during validation rather than during mutation.

5.5.2 Collection Property

A collection property may take the form of an *indexed property* or a *list property*. The base type of an indexed property may be either a primitive type or a reference type, while that of a list property must be a reference type.

A collection consists of a group of collection items. If one of the collection items can represent a nillable element declaration, setting a collection item to `null` is semantically equivalent to inserting a nil element, `xsi:nil="true"`, into the collection property. If none of the collection items can ever represent a nillable element declaration, setting a collection item to `null` is the semantic equivalent of removing an optional element from the collection property.

5.5.2.1 Indexed Property

This property follows the indexed property design pattern for a multi-valued property from the JavaBean specification. An indexed property *prop* with base type *Type* is realized by the five methods

regardless of whether *Type* is a primitive type or a reference type. *Id* is computed from `prop` as it was defined in simple property. An array item is a specialization of the collection item abstraction introduced in the collection property overview.

- `get`*Id*`()`
  The array `getter` method returns an array containing the property's value. If the property's value has not set, then `null` is returned.
- `set`*Id*`(`*Type*` [])`
  The `array setter` method defines the property's set value. If the argument itself is `null` then the property's set value, if any, is discarded. If the argument is not `null` and `TypeConstraint` validation is enabled [5] then the sequence of values in the array are first validated by applying the property's predicate, which may throw a `TypeConstraintException`. If the `TypeConstraintException` is thrown, the property retains the value it had prior to the `set` method invocation. The property's value is only modified after the `TypeConstraint` validation step.
- `set`*Id*`(int, `*Type*`)`
  The indexed `set`ter method allows one to set a value within the array. The runtime exception `j a v a` .

he current array bounds. If
straint validation is

[5], the value is validated against the property's predicate, which may throw an unchecked `TypeConstraintException`. If

setting a property v is th of the specification.

to the same value it had before the invocation of the indexed `setter` method. When the array item represents a nillable element declaration and the indexed setter value parameter is null, it is semantically equivalent to inserting a nil element into the array.

- `get`*`Id`*`(int)`
  The indexed `get`ter method returns a single element from the array. The runtime exception `java.lang.ArrayIndexOutOfBoundsException` may be thrown if the index is used outside the current array bounds. In order to change the size of the array, you must use the array set method to set a new (or updated) array.

- `get`*`Id`*`Length()`
  The indexed length method returns the length of the array. This method enables you to iterate over all the items within the indexed property using the indexed mutators exclusively. Exclusive use of indexed mutators and this method enable you to avoid the allocation overhead associated with array `getter` and `setter` methods.

The arrays returned and taken by these methods are not part of the content object's state. When an array `getter` method is invoked, it creates a new array to hold the returned values. Similarly, when the corresponding array `setter` method is invoked, it copies the values from the argument array.

To test whether an indexed property has a set value, invoke its `array getter` method and check that the result is not `null`. To discard an indexed property's set value, invoke its array `setter` method with an argument of `null`.

See the customization attribute `collectionType` in Section 7.5, "`<globalBindings>` Declaration" and Section 7.8, "`<property>` Declaration" on how to enable the generation of indexed property methods for a collection property.

Example

In the purchase order schema, we have the following repeating element occurrence of element item within `complexType` Items.

```
<xs:complexType name="Items">
    <xs:sequence>
        <xs:element name="item" minOccurs="1"maxOccurs="unbounded">
            <xs:complexType>...</xs:complexType>
        </xs:element>
</xs:complexType>
```

The content specification of this element type could be bound to an array property realized by these five methods:

```
public Items.ItemType[] getItem();
public void setItem(Items.ItemType[] value);
public void setItem(int index, Items.ItemType value);
public Items.ItemType getItem(int index);
public int getItemLength();
```

5.5.2.2 List Property

A list property *prop* with base type *Type* is realized by the method where `List`

```
public List<Type> getId();
```

is the interface `java.util.List`, *`Id`* is defined as above. If base type is a primitive type, the appropriate wrapper class is used in its place.

- The `get` method returns an object that implements the `List<Type>` interface, is mutable, and contains the values of type *Type* that constitute the property's value. If the property does not have a set value or a schema default value, a zero length `java.util.List` instance is returned.

The `List` returned by the `get` method is a component of the content object's state. Modifications made to this list will, in effect, be modifications to the content object. If `TypeConstraint` validation is enabled, the list's mutation methods apply the property's predicate to any non-`null` value before adding that value to the list or replacing an existing element's value with that value; the predicate may throw a `TypeConstraintException`. The collection property overview discussion on setting a collection item to null specifies the meaning of inserting a null into a List.

The `unset` method introduced in Section 5.5.4, "`isSet` Property Modifier," on page 61 enables one to discard the set value for a List property.

Design Note – There is no setter method for a List property. The getter returns the List by reference. An item can be added to the List returned by the getter method using an appropriate method defined on `java.util.List`. Rationale for this design in JAXB 1.0 was to enable the implementation to wrapper the list and be able to perform checks as content was added or removed from the List.

Example

The content specification of the `item` element type could alternatively be bound to a list property realized by one method:

```
public List<Item> getItem();
```

The list returned by the `getItem` method would be guaranteed only to contain instances of the `Item` class. As before, its length would be checked only during validation, since the requirement that there be at least one `item` in an element instance of complex type definition `Items` is a structural constraint rather than a type constraint.

5.5.3 Constant Property

An attribute use named *prop* with a schema specified fixed value can be bound to a Java constant value. *Id* is computed from `prop` as it was defined in simple

```
static final public Type ID = <fixedValue>;
```

property. The value of the fixed attribute of the attribute use provides the *<fixedValue>* constant value.

The binding customization attribute `fixedAttributeToConstantProperty` enables this binding style. Section 7.5, "<globalBindings> Declaration" and Section 7.8, "<property> Declaration" describe how to use this attribute.

5.5.4 `isSet` Property Modifier

This optional modifier augments a modifiable property to enable the manipulation of the property's value as a *set value* or a *defaulted value*. Since this functionality is above and beyond the typical JavaBean pattern for a property, the method(s) associated with this modifier are not generated by default. Chapter 7, "Customizing XML Schema to Java Representation Binding" describes how to enable this customization using the `generateIsSetMethod` attribute.

The method signatures for the `isSet` property modifier are the following:

```
public boolean isSetId();
```

where *Id* is defined as it was for simple and collection property.

- The `isSet` method returns `true` if the property has been set during unmarshalling or by invocation of the mutation method `setId` with a non-`null` value. [6]

  To aid the understanding of what `isSet` method implies, note that the unmarshalling process only unmarshals *set values* into XML content.

A list property and a simple property with a non-reference base type require an additional method to enable you to discard the *set value* for a property:

```
public void unsetId();
```

- The `unset` method marks the property as having no *set value*. A subsequent call to `getId` method returns the schema-specified default if it existed; otherwise, it returns the Java default initial value for `Type`.

All other property kinds rely on the invocation of their set method with a value of null to discard the set value of its property. Since this is not possible for primitive types or a List property, the additional method is generated for these cases illustrate the method invocations that result in transitions between the possible states of a JAXB property value.

6. A Java application usually does not need to distinguish between the absence of a element from the infoset and when the element occurred with nil content. Thus, in the interest of simplifying the generated API, methods were not provided to distinguish between the two. Two annotation elements @XmlElement.required and @XmlElement.nillable allow a null value to be marshalled as an empty or nillable element.

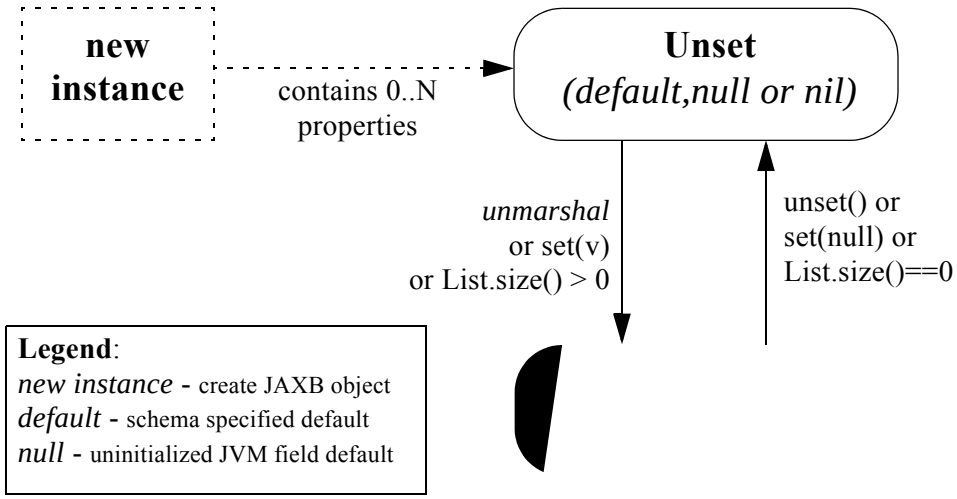


Figure 5.1 States of a Property Value

Example

In the purchase order schema, the `partNum` attribute of the element `item`'s anonymous complex type is declared:

```
<xs:attribute name="partNum" type = "SKU" use="required"/>
```

This attribute could be bound to a `isSet` simple property realized by these four methods:

```
public String getPartNum();
public void setPartNum(String skuValue);
public boolean isSetPartNum();
public void unsetPartNum();
```

It is legal to invoke the `unsetPartNum` method even though the attribute's `use` is "`required`" in the XML Schema. That the attribute actually has a value is a local structural constraint rather than a type constraint, so it is checked during validation rather than during mutation.

5.5.5 Element Property

This property pattern enables the dynamic association of an element name for a JAXB property. Typically, the element name is statically associated with a

JAXB property based on the schema's element name. Element substitution groups and wildcard content allow an XML document author to use Element names that were not statically specified in the content model of the schema. To support these extensibility features, an application uses element property setters/ getters to dynamically introduce element names at runtime.

The method signatures for the `Element` property pattern are the following:

```
public void setId (JAXBElement<? extends Type> value);
public JAXBElement<? extends Type> getId();
```

where *`Id`* and *`Type`* are defined as they were for simple and collection property. The fully qualified Java name for *`JAXBElement <T>`* is *`javax.xml.bind.JAXBElement<T>`*. The generic types in the method signatures are expressed as a bounded wildcard to support element substitution group, see details in Section 6.7, "Element Declaration".

5.5.6 Property Summary

The following core properties have been defined:

- Simple property - JavaBean design pattern for single value property
- Indexed property - JavaBean design pattern for multi-valued property
- List property - Leverages java.util.Collection
- Constant property

The methods generated for these four core property kinds are sufficient for most applications. Configuration-level binding schema declarations enable an application to request finer control than provided by the core properties. For example, the `isSet` property modifier enables an application to determine if a property's value is set or not.

5.6 Java Element Representation

Based on rationale and criteria described in Section 6.7, "Element Declaration," on page 104, the schema compiler binds an element declaration to a Java instance that implements `javax.xml.bind.JAXBElement<T>`. `JAXBElement<T>` class provides access to the basic properties of an XML element: its name, the value of the element's datatype, and whether the

element's content model is set to nil, i.e. `xsi:nil="true"`. Optional properties for an Xml element that corresponds to an element declaration from a known schema include the element declaration's declared type and scope.

The enhanced, default binding for an element declaration only generates a element instance factory method and is described in Section 5.6.1, "Named Java Element instance".[7] The customized binding that generates a schema-dervied Element class for an element declaration is described in Section 5.6.2, "Java Element Class".

5.6.1 Named Java Element instance

Based on the normative binding details described in Section 6.7.1, "Bind to `JAXBElement<T>` Instance," on page 106, the schema compiler binds an element declaration to an element instance factory method.

The following is a generic element factory signature.

```
package elementDeclarationTargetNamespace;
class ObjectFactory {
    javax.xml.bind.JAXBElement<ElementType>
                                createElementName(ElementType value);
}
```

The element factory method enables an application to work with elements without having to directly know the precise `javax.xml.namespace.QName`. The element factory method abstraction sets the Xml element name with the Java representation of the element, thus shielding the JAXB user from the complexities of manipulating namespaces and QNames.

CODE EXAMPLE 5-1 Binding of global element declaration to element factory

```
<xs:schema targetNamespace="a" xmlns:a="a"/>
<xs:element name="Foo" type="xsd:int"/>
```

7. The exception case is that an element declaration with an anonymous type definition is bound to a schema-derived value class by default as described in Section 6.7.3, "Binding of an anonymous complex type definition".

```
class ObjectFactory {
    // returns JAXBElement with its name set to QName("a", "Foo").
    JAXBElement<Integer> createFoo(Integer value);
}
```

5.6.2 Java Element Class

Based on criteria to be identified in Section 6.7.2, "Bind to Element Class," on page 108, the schema compiler binds an element declaration to a Java element class. An element class is defined in terms of the properties of the "Element Declaration Schema Component" on page 349 as follows:

- An element class name is generated from the element declaration's name using the XML Name to Java identifier name mapping algorithm specified in Section D.2, "The Name to Identifier Mapping Algorithm," on page 333.
- Scope of element class
  - Global element declarations are declared in package scope.
  - By default, local element declarations occur in the scope of the first ancestor complex type definition that contains the declaration. The schema customization <jaxb:globalBindings localScoping="toplevel"/>, specified in Section 7.5.1, disables the generation of schema-derived nested classes and all classes are generated as toplevel classes.
- Each generated Element class must extend the Java class `javax.xml.bind.JAXBElement<T>`. The type T of the `JAXBElement<T>` is derived from the element declaration's type. Anonymous type definition binding is a special case that is specified in Section 6.7.3, "Binding of an anonymous complex type definition," on page 110
- A factory method is generated in the package's `ObjectFactory` class introduced in Section 5.2, "Java Package." The factory method returns JAXBElement<T>. The factory method has one parameter that is of type `T`. The name of the factory method is generated by concatenating the following components:
  - The string constant `create`.
  - If the Java element class is nested within a value class, then the concatenation of all outer Java class names.

- The *name* of the Java value class.

The returned instance has the Xml Element name property set to the QName representing the element declaration's name.

For example, a Java element class named `Foo` that is nested within Java value class `Bar` would have the following factory method generated in the containing Java package's `ObjectFactory` class:

```
JAXBElement<Integer> createBarFoo(Integer value)
```

- A public no-arg constructor is generated.
  The constructor must set the appropriate Xml element name, just as the element factory method does.
- The Java element representation extends JAXBElement<`T`> class, its properties provide the capability to manipulate
  - the value of the element's content
    Xml Schema's type substitution capability is enabled by this property.
  - whether the element's content model is `nil`

Example:

Given a complex type definition with mixed content:

```
<xs:complexType name="AComplexType" mixed="true">[8]
    <xs:sequence>
        <xs:element name="ASimpleElement" type="xs:int"/>[9]
    </xs:sequence>
</xs:complexType>
```

8. Section 6.12.4, "Bind mixed content," on page 143 describes why <ASimpleElement> element is bound to a Java Element representation.

9. Assume a customization that binds this local element declaration to an element class. By default, this local declaration binds to a element instance factory returning JAXBElement<Integer>

Its Java representation:

```
public value class AComplexType {
    public class ASimpleElement extends
                          javax.xml.bind.JAXBElement<Integer> {
    }
    ...
};
class ObjectFactory {
    AComplexType createAComplexType();
    JAXBElement<Integer>
           createAComplexTypeASimpleElement(Integer value);
    ...
}
```

5.6.3 Java Element Representation Summary

Element declaration binding evolved in JAXB v2.0 to support XML Schema type substitution. The following diagrams illustrate the binding changes for the following schema fragment:

```
<xs:element name="foo" type="fooType"/>
```

Figure 5.1 JAXB 1.0 : isA Relationship between generated element interface and its type

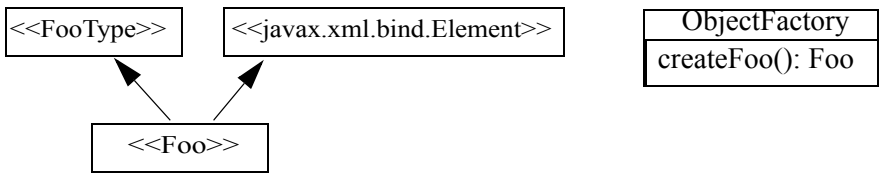


Figure 5.2 JAXB 2.0: hasA Relationship between element instance and its type

as described in Section 5.6.1, “Named Java Element instance”

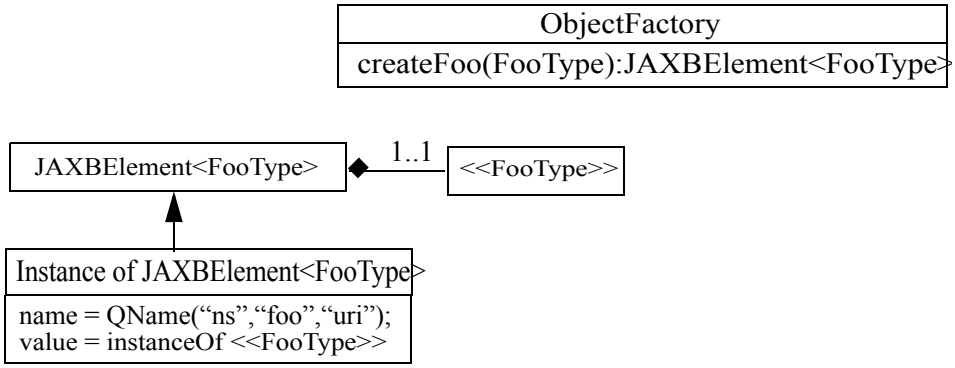
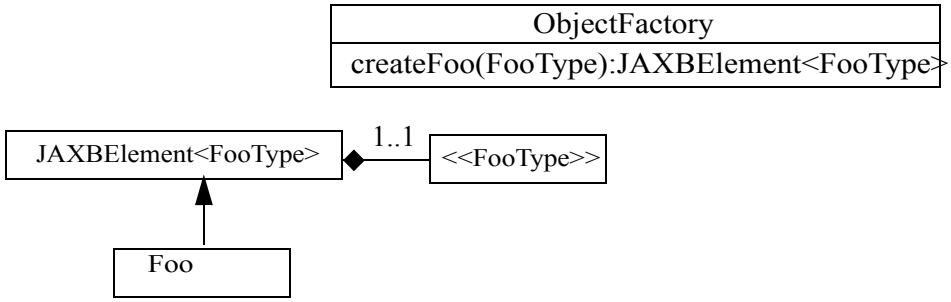


Figure 5.3 JAXB 2.0: hasA Relationship between generated element class and its type as described in Section 5.6.2, “Java Element Class”



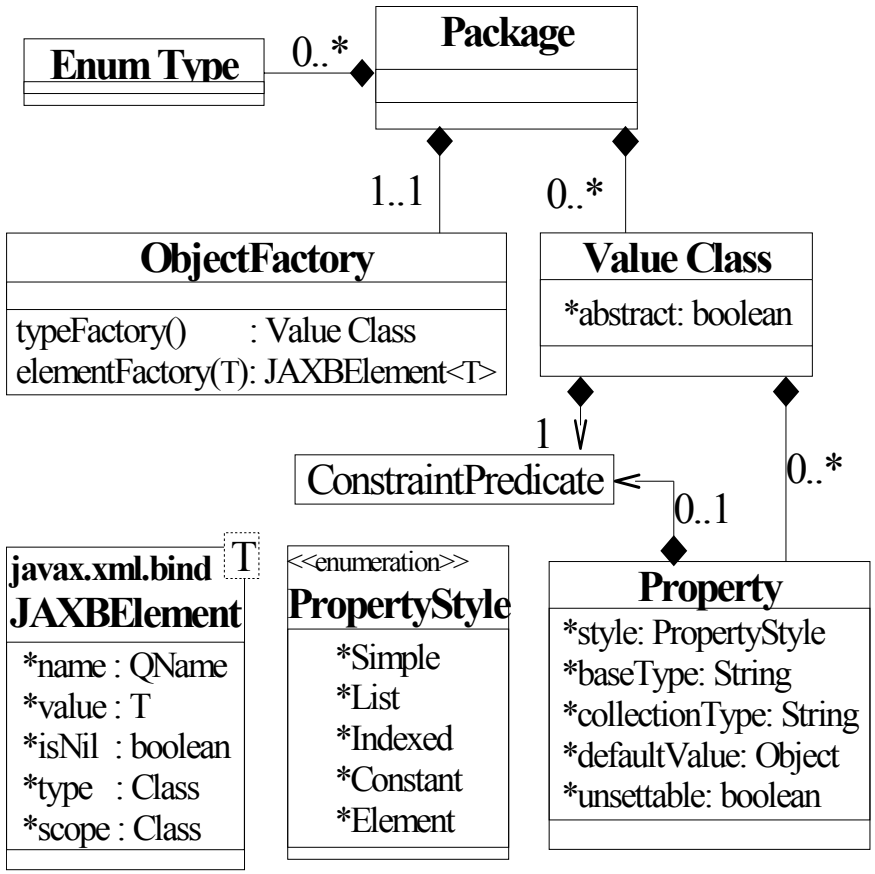
While a JAXB v1.0 Element interface implemented its type’s interface, a JAXB v2.0 Element instance has a composition relationship to the value of the element declaration’s type, accessible via the `javax.xml.bind.JAXBElement<T>` property `Value`. This change reflects the relationship that type substitution allows an element declaration to be associated with many different datatypes, not just the datatype that was defined statically within the schema.

An added benefit to the default binding change is to reduce the overhead associated with always generating Java Element classes for every global element declaration.In JAXB 1.0, an interface was generated for every complex type definition and global element declaration. In JAXB 2.0, a value class is generated for every complex type definition and only a factory method needs to be generated for each global element declaration.

5.7 Summary

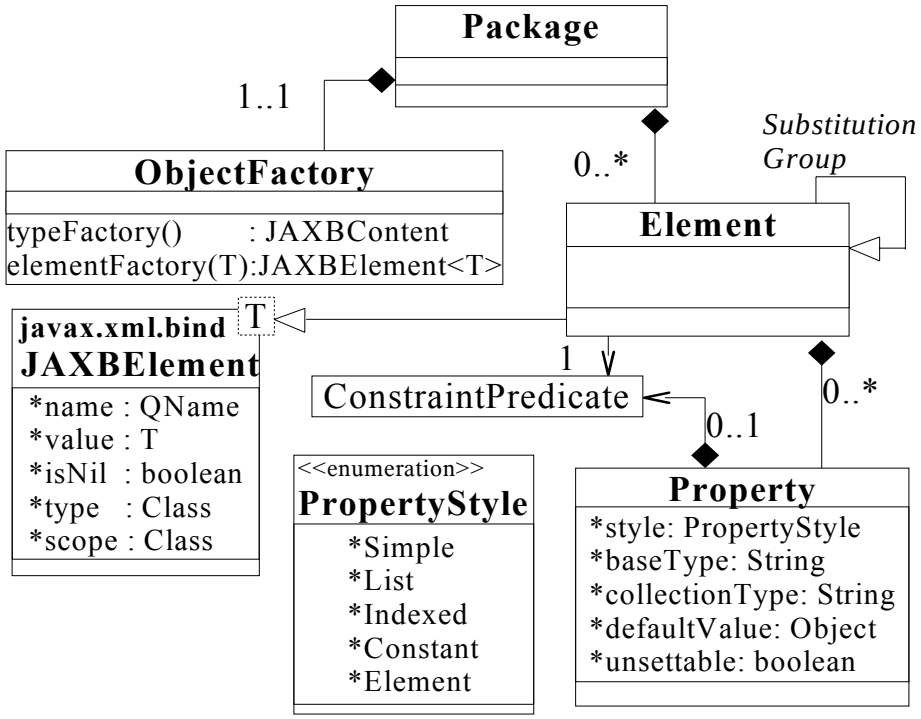
The composition and relationships between the Java components introduced in this section are reflected in the following diagram.

Figure 5.1 UML diagram of Java Representation[10]



[10] See next figure for default binding for anonymous type definition.

Figure 5.2 UML diagram when xs:element is bound to schema-derived Element class



See also Table 6-14, "Summarize default XSD to Java binding for Figure 5.4 and Figure 5.5".

CHAPTER 6

BINDING XML SCHEMA TO JAVA REPRESENTATIONS

This chapter describes binding of XML schema components to Java representations. The default binding is identified in this chapter and the next chapter specifies the customizations that override default binding.

6.1 Overview

The abstract model described in [XSD Part 1] is used to discuss the default binding of each schema component type. Each schema component is described as a list of properties and their semantics. References to properties of a schema component as defined in [XSD Part 1] are denoted using the notation *{schema property}* throughout this section. References to properties of information items as defined in [XML-Infoset] are denoted by the notation **[property]**.

All JAXB implementations are required to implement the default bindings specified in this chapter. However, users and JAXB implementors can use the global configuration capabilities of the custom binding mechanism to override the defaults in a portable manner.

For each binding of a schema component to its Java representation, there is a description of Java mapping annotation(s), described in Chapter 8, “Java Types To XML“, to be generated with the Java representation. The standardization of these mapping annotations assist in specifying the portability of a schema-derived JAXB-annotated classes. All JAXB implementations are required to be able to unmarshal/marshal another implementation’s schema-derived Java value classes by interpreting the specified mapping annotations. Note that each mapping annotation is described independent of whether it is the default mapping or a customized mapping, JAXB implementations are allowed to

optimize away redundant mapping annotations that are the default mapping annotation.

Design Note – Note that the mapping annotations generated on the schema derived classes do not capture all aspects from the original schema. A schema generated from the mapping annotations of the schema derived classes differs from the original schema used to generate the schema-derived classes. The original schema is more precise for validation purposes than the one generated from the schema-derived classes.

All examples are non-normative. Note that in the examples, the schema-derived code does not list all required mapping annotations. In the interest of being terse, only the mapping annotations directly connected to the schema component being discussed are listed in its example.

6.2 Simple Type Definition

A schema component using a simple type definition typically binds to a Java property. Since there are different kinds of such schema components, the following Java property attributes (common to the schema components) are specified here and include:

- base type
- collection type if any
- predicate

The rest of the Java property attributes are specified in the schema component using the simple type definition.

While not necessary to perform by default, this section illustrates how a simple type definition is bound to a JAXB mapped class. This binding is necessary to preserve a simple type definition referred to by `xsi:type` attribute in an Xml instance document. See Section 7.5.1, “Usage“for the customization that enables this binding.

6.2.1 Type Categorization

The simple type definitions can be categorized as:

- schema built-in datatypes [XSD PART2]
- user-derived datatypes

Conceptually, there is no difference between the two. A schema built-in datatype can be a primitive datatype. But it can also, like a user-derived datatype, be derived from a schema built-in datatype. Hence no distinction is made between the schema built-in and user-derived datatypes.

The specification of simple type definitions is based on the abstract model described in Section 4.1, “Simple Type Definition” [XSD PART2]. The abstract model defines three varieties of simple type definitions: atomic, list, union. The Java property attributes for each of these are described next.

6.2.2 Atomic Datatype

If an atomic datatype has been derived by restriction using an “enumeration” facet, the Java property attributes are defined by Section 6.2.3, “Enum Type.” Otherwise they are defined as described here.

The base type is derived upon the XML built-in type hierarchy [XSD PART2, Section 3] reproduced below.

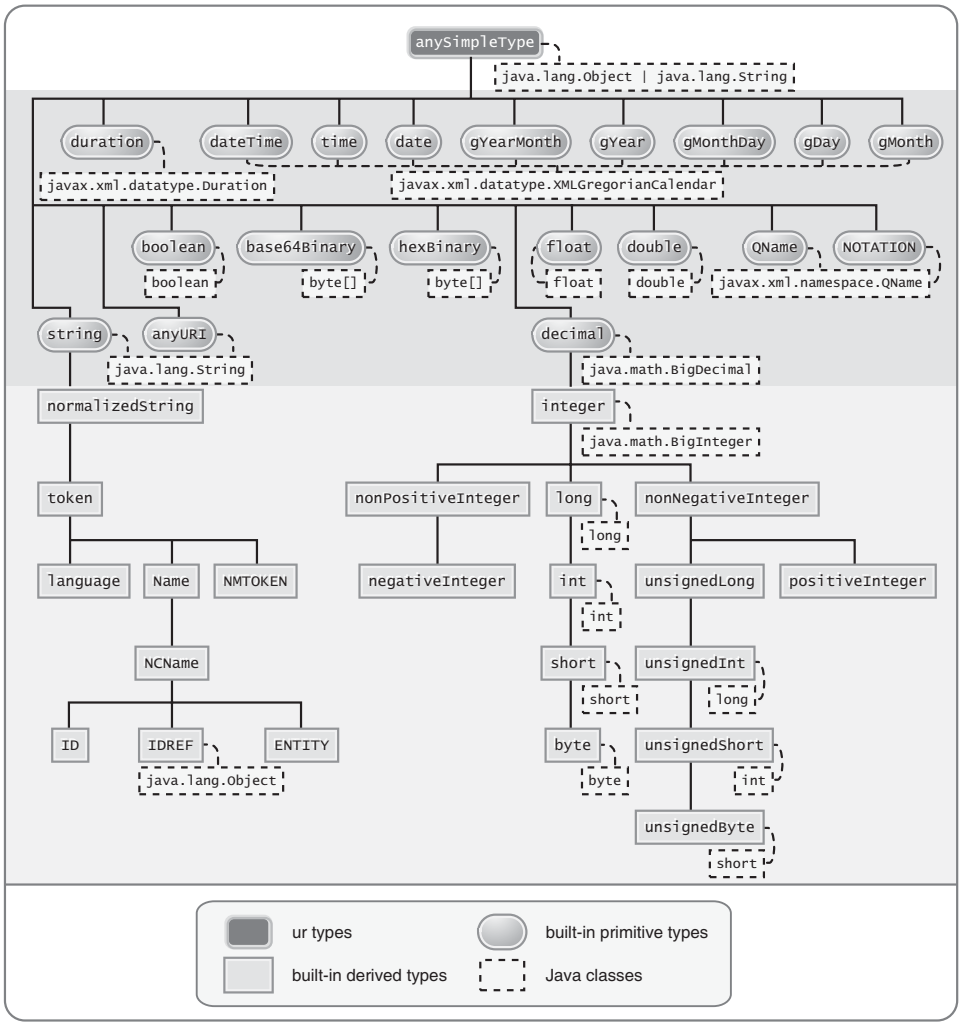


Figure 6.1 XML Built-In Type Hierarchy

The above diagram is the same as the one in [XSD PART2] except for the following:

- Only schema built-in atomic datatypes derived by restriction have been shown.

- The schema built-in atomic datatypes have been annotated with Java data types from the "Java Mapping for XML Schema Built-in Types" table below.

Design Note – xs:anyURI is not bound to java.net.URI by default since not all possible values of xs:anyURI can be passed to the java.net.URI constructor. Using a global JAXB customization described in Section 7.9, "`<javaType>` Declaration", a JAXB user can override the default mapping to map xs:anyURI to java.net.URI.

The following is a mapping for subset of the XML schema built-in data types to Java data types. This table is used to specify the base type later.

Table 6-1 Java Mapping for XML Schema Built-in Types

XML Schema Datatype	Java Datatype
xsd:string	**java.lang.String**
xsd:integer	**java.math.BigInteger**
xsd:int	**int**
xsd.long	**long**
xsd:short	**short**
xsd:decimal	**java.math.BigDecimal**
xsd:float	**float**
xsd:double	**double**
xsd:boolean	**boolean**
xsd:byte	**byte**
xsd:QName	**javax.xml.namespace.QName** [1]
xsd:dateTime	javax.xml.datatype.XMLGregorianCalendar [1]
xsd:base64Binary	**byte[]**
xsd:hexBinary	**byte[]**
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	javax.xml.datatype.XMLGregorianCalendar [1]
xsd:date	javax.xml.datatype.XMLGregorianCalendar [1]
xsd:g*	javax.xml.datatype.XMLGregorianCalendar [1]

Table 6-1 Java Mapping for XML Schema Built-in Types

XML Schema Datatype	Java Datatype
xsd:anySimpleType (for xsd:element of this type)[a]	java.lang.Object
xsd:anySimpleType (for xsd:attribute of this type)	java.lang.String
xsd:duration	javax.xml.datatype.Duration [1]
xsd:NOTATION	javax.xml.namespace.QName [1]

a. enable type substitution for element of xsd:anySimpleType

[1.] JAXP 1.3 defines package `javax.xml.datatype` and `javax.xml.namespace`

The base type is determined as follows:

1. Map by value space bounding facets
   If the simple type derives from or is `xsd:integer` and has either a constraining lower and/or upper bounds facet(s) or totalDigits facet, check if the following optimized binding is possible:
   - If the simple type derives from or is `xsd:short`, `xsd:byte` or `xsd:unsignedByte`, go to step 2.
   - If the value space for the simple type is representable in the range of `java.lang.Integer.MIN_VALUE` and `java.lang.Integer.MAX_VALUE`, map to java primitive type, `int`.
   - If the value space for the simple type is representable in the range of `java.lang.Long.MIN_VALUE` and `java.lang.Long.MAX_VALUE`, map to java primitive type, `long`.
   - Else go to step 2.
2. **Map by datatype**
   If a mapping is defined for the simple type in Table 6.1, the base type defaults to its defined Java datatype.
3. **Map by base datatype**
   Otherwise, the base type must be the result obtained by repeating the step 1 and 2 using the *{base type definition}*. For schema datatypes derived by restriction, the *{base type definition}* represents the simple type definition from which it is derived. Therefore, repeating step 1

with *{base type definition}* essentially walks up the XML Schema built-in type hierarchy until a simple type definition which is mapped to a Java datatype is found.

The Java property predicate must be as specified in "Simple Type Definition Validation Rules," Section 4.1.4[XSD PART2].

Example:

The following schema fragment (taken from Section 4.3.1, "Length" [XSD PART2]):

```
<xs:simpleType name="productCode">
    <xs:restriction base="xs:string">
        <xs:length value="8" fixed="true"/>
    </xs:restriction>
</xs:simpleType>
```

The facet "length" constrains the length of a product code (represented by `productCode`) to 8 characters (see section 4.3.1 [XSD PART2] for details).

The Java property attributes corresponding to the above schema fragment are:

- There is no Java datatype mapping for `productCode`. So the Java datatype is determined by walking up the built-in type hierarchy.
- The `{base type definition}` of `productCode` is `xs:string`. `xs:string` is mapped to `java.lang.String` (as indicated in the table, and assuming no customization). Therefore, `productCode` is mapped to the Java datatype `java.lang.String`.
- The predicate enforces the constraints on the length.

6.2.2.1 Notation

Given that the value space of `xsd:NOTATION` is the set of `xsd:QName`, bind `xsd:NOTATION` type to `javax.xml.namespace.QName`.

For example, the following schema:

```
<xs:schema targetNamespace="http://e.org" xmlns:e="http://e.org"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:notation name="jpeg" public="image/jpeg" system="jpeg.exe"/>
<xs:notation name="png"  public="image/png" system="png.exe"/>
<xs:simpleType name="pictureType">
    <xs:restriction base="xs:NOTATION">
        <xs:enumeration value="e:jpeg"/>
        <xs:enumeration value="e:png"/>
    </xs:restriction>
</xs:simpleType>
<xs:complexType name="Picture">
    <xs:simpleContent>
        <xs:extension base="xs:hexBinary">
            <xs:attribute name="format" type="e:pictureType"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
</xs:schema>
```

is mapped to the following Java code:

```
package org.e;
import javax.xml.namespace.QName;
public class Picture {
    void setValue(byte[] value) {...}
    byte[] getValue() {...}
    void setFormat(QName value){...}
    QName getFormat(){...}
}
```

With the following usage scenario:

```
Picture pic = ...;
pic.setFormat(new QName("http://e.org","jpeg"));
```

6.2.2.2 Bind to a JAXB mapped class

By default, a named simple type definition is not bound to a Java class. This binding is only necessary to enable the precise type of an `xsi:type` substitution to be preserved as described in Section 6.7.4.2, “Type Substitution of a Simple Type Definition“. This binding is enabled via the global binding customization attribute *@mapSimpleTypeDef* specified in Section 7.5.1, “Usage“.

The binding of a named simple type definition to a Java value class is based on the abstract model properties in Section F.1.1, "Simple Type Definition Schema Component," on page 347. The Java value class must be defined as specified here, unless the ref attribute is specified on the `<jaxb:class>` declaration, in which case the schema compiler will simply assume that the nominated class is already bound to this simple type.

- **name:** name is the Java identifier obtained by mapping the XML name `{name}` using the name mapping algorithm, specified in Section D.2, "The Name to Identifier Mapping Algorithm," on page 333. Note that anonymous simple type definition's are never bound to a Java value class.
- **package:**
  - The schema-derived Java value class is generated into the Java package that represents the binding of *{target namespace}*
- **outer class name:**
  - There is no outer class name for a global simple type definition.
- **base class:**
  - Due to a constraint specified for @XmlValue in Section 8, this class can not extend any other class. The derivation by restriction hierarchy for simple type definitions can not be captured in the schema-derived Java value class.
- **value property:**
  - Same as the binding of simple content in Section 6.3.2.1, "Simple Content Binding," on page 95 to an @XmlValue annotated JAXB property.

The next two examples illustrate the binding of a simple type definition to a Java value class when the appropriate JAXB schema customization is enabled.

CODE EXAMPLE 6-1 Simple type definition

```
<xs:simpleType name="productCode">
<xs:restriction base="xs:string">
<xs:length value="8" fixed="true"/>
</xs:restriction>
</xs:simpleType>
```

CODE EXAMPLE 6-2 Binding of Section CODE EXAMPLE 6-1, "Simple type definition"

```
@XmlType(name="productCode")
public class ProductCode {
@XmlValue
String getValue();
void setValue(String value);
}
```

6.2.2.3 Annotations for standard XML datatypes

By default, a schema-derived JAXB property bound from one of the following standard XML datatypes is annotated with the specified mapping annotation.

Schema Type	JAXB Property Annotation
xsd:ID	@XmlID
xsd:IDREF	@XmlIDREF
ref:swaRef	@XmlAttachmentRef

Note that JAXB schema customizations could override these default binding.

6.2.3 Enum Type

The default mapping for a named atomic type that is derived by restriction with enumeration facet(s) and whose restriction base type (represented by *{base type definition}*) is `xs:String`[1] or derived from it is mapped to an enum type. The **`[typesafeEnumBase]`** attribute customization described in Section 7.5, "`<globalBindings>` Declaration," enables global configuration to alter what Xml built-in datatypes are bound by default to an enum type. An anonymous simple type definition is never bound to an enum class by default, but it can be customized as described in Section 7.10, "`<typesafeEnum>` Declaration" to bind to an enum type.

[1] Exception cases that do not bind to enum type: when the base type is or derives from `xs:ID` and `xs:IDREF`. Rationale for not binding these type definitions to an enum type is in Section 7.10.5, "Customizable Schema Elements".

6.2.3.1 Example binding

An example is provided first followed by a more formal specification.

XML Schema fragment:

```
<xs:simpleType name="USState">
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="AK"/>
        <xs:enumeration value="AL"/>
    </xs:restriction>
</xs:simpleType>
```

The corresponding enum type binding is:

```
public enum USState {
    AK, AL;
    public String value() { return name(); }
    public static USState fromValue(String value) { ... }
};
```

6.2.3.2 Enum type binding

The characteristics of an *enum type* are derived in terms of the properties of the "Simple Type Definition Schema Component" on page 347 as follows:

The enum type binding is defined as follows:

- **name:** The default name of the enum type, *enumType,* is computed by applying the XML Name to Java identifier mapping algorithm to the *{name}* of the simple type definition. There is no mechanism to derive a name for an anonymous simple type definition, the customization must provide the **name**.
- **package name:** The package name is determined from the *{targetname space}* of the schema that directly contains the simple type definition.
- **outer class name:**
    - There is no **outer class name** for a global simple type definition.
    - There is no **outer class name** when schema customization, **[jaxb:globalBindings]** *@localScoping*, specified in Section 7.5.1, has a value of *toplevel*.
    - The **outer class name** for an anonymous simple type definition is computed by traversing up the anonymous simple type definition's ancestor tree until the first ancestor is found that is:

- an XML component that is mapped to a Java value class, the **outer class name** is composed of the concatenation of this Java value class's **outer class name**, "."**,** and its **name**.
- a global declaration or definition is reached. There is no **outer class name** for this case.

- **enum constants**: Specified in next section.

Note that since a Java enum type is essentially a final class, it is not possible for it to be subclassed. Thus, any derivations of a simple type definition bound to an enum type can not be captured by an equivalent Java inheritance relationship.

The schema-derived enum is annotated, either explicitly or by default mapping annotations, with the mapping annotation @XmlEnum, specified in Section 8. The @XmlEnum annotation elements are derived in terms of the abstract model properties for a simple type definition summarized in Section F.1.1, “Simple Type Definition Schema Component,” on page 347 as follows:

Table 6-1 Annotate enum type with @XmlEnum element-value pairs

@XmlEnum element	@XmlEnum value
name	simple type definition's {name}
namespace	{target namespace}
value	the java type binding of the simple type definition’s *{base type definition}*

6.2.3.3 Enum Constant

An enum constant is derived for each enumeration facet of the atomic type definition. The characteristics of an *enum constant* of the enum type are derived in terms of the properties of the “Enumeration Facet Schema Component” on page 348 as follows:

- **name:** The name is either specified via customization, `jaxb:typesafeEnumMember` described in Section 7.10.1, “Usage“, or the name is computed as specified in Section 6.2.3.4, “XML Enumvalue-to-Java Identifier Mapping“.
- **type:** The Java type binding of the simple type definition’s {*base_type_definition*}.
- **value**: The conversion of string *{value}* to **type**. **Value** is manipulated via the following generated enum type methods:

```
public        type          value();
public static enumTypeName  fromValue(type value);
```

To assist an application in manipulating the enum constants that comprise an enum type, all enum types have the following two implicitly declared static methods as specified in Section 8.9 in [JLS3]. The enum type’s static method `values()` returns an array of all enum constants. The static method `valueOf(String name)` returns the enum constant represented by the name parameter.

6.2.3.4 XML Enumvalue-to-Java Identifier Mapping

The default name for the enum constant is based on mapping of the XML enumeration value to a Java identifier as described below.

The XML enumeration value *{value}* is mapped to a Java Identifier using the algorithm specified in Section D.3, "Deriving a legal Java identifier from an enum facet value".If there is a collision among the generated constant fields **name** or if it is not possible to generate a legal Java identifier for one or more of the generated constant field names, see Section 7.5.5, "@typesafeEnumMemberName"for customization options to resolve this error case.

6.2.3.5 Enum Constant Name differs from its Value

For all cases where there exist at least one enumeration constant name that is not the same as the enumeration constant's value, the generated enum type must have a final value field that is set by the enum type's constructor. The code generation template is the following:

CODE EXAMPLE 6-1 At least one enum constant name differs from its value.

```
public enum enumType {
    EnumConstantName1(EnumConstantValue1),
    ...
    EnumConstantNameX(EnumConstantValueX);
    public EnumConstantValueType value() { return value;}
    public static enumType fromValue(EnumConstantValueType val)
        { ... }

    final private EnumConstantValueType value;
    private enumType(EnumConstantValueType value) {
        this.value = value;
    }
}
```

CODE EXAMPLE 6-2 Code template when enum constant name is same as its enum constant value.

```
public enum enumType {
    EnumConstantName1, ..., EnumConstantNameX;
    public String[2] value() { return name();}
    public static enumType fromValue(String[1] value) { ... }
}
```

[2] Note for this case, the *enumConstantValueType* is always `java.lang.String`.

The schema-derived enum constant is annotated, either explicitly or by default mapping annotations, with the mapping annotation specified in Section 8. The `@XmlEnumValue` annotation elements are derived in terms of the abstract model properties for a enumerated facet summarized in Section F.1.2, "Enumeration Facet Schema Component," on page 348 as follows:

Table 6-1 Annotate enum constant with @XmlEnumValue element-value pairs

@XmlEnumValue element	@XmlEnumValue value
value	Enumeration facet's {value}

CODE EXAMPLE 6-3 Schema-derived enum type when enumeration facet's value does not match enum constant name.

Given following schema fragment:

```
<xs:simpleType name="Coin">
    <!-- Assume jaxb customization that binds Coin to an enumType-->
    <xs:restriction base="xs:int">
     <!--Assume jaxb customization specifying enumConstantName-->
        <xs:enumeration value="1"/> <!-- name="penny"-->
        <xs:enumeration value="5"/> <!-- name="nickel"-->
        <xs:enumeration value="10"/><!-- name="dime"-->
        <xs:enumeration value="25"/><!-- name="quarter-->
    </xs:restriction>
</xs:simpleType>
```

Schema-derived enum type:

```
@XmlEnum(value="java.lang.Integer.class")
public enum Coin {
    @XmlEnumValue("1") PENNY(1),
    @XmlEnumValue("5") NICKEL(5),
    @XmlEnumValue("10") DIME(10),
    @XmlEnumValue("25") QUARTER(25);
    public int value(){ return value; }
    public static Coin fromValue(int value) { ... }

    final private Integer value;
    Coin(int value) { this.value = value; }
}
```

6.2.4 List

A list simple type definition can only contain list items of atomic or union datatypes. The item type within the list is represented by the schema property *{item type definition}*.

The Java property attributes for a list simple type definition are:

- The *base type* is derived from the *{item type definition}* as follows. If the Java datatype for *{item type definition}* is a Java primitive type, then the base type is the wrapper class for the Java primitive type. Otherwise, the Java datatype is derived from the XML datatype as specified in Section 6.2.2, "Atomic Datatype" and Section 6.2.3, "Enum Type."
- The *collection type* defaults to an implementation of java.util.List. Note that this specification does not specify the default implementation for the interface java.util.List, it is implementation dependent.
- The *predicate* is derived from the "Simple Type Definition Validation Rules," in section 4.1.4,[XSD PART2].

Example:

For the following schema fragment:

```
<xs:simpleType name="xs:USStateList">
   <xs:list itemType="xs:string"/>
</xs:simpleType>
```

The corresponding Java property attributes are:

- The *base type* is derived from *{item type definition}* which is XML datatype, *"xs:string"*, thus the Java datatype is java.util.String as specified in Section Table 6-1, "Java Mapping for XML Schema Built-in Types."
- The *collection type* defaults to an implementation of java.util.List.
- The *predicate* only allows instances of *base type* to be inserted into the list. When failfast check is being performed[3], the list's mutation methods apply the property's predicate to any non-`null` value before adding that value to the list or replacing an existing element's value with that value; the predicate may throw a `TypeConstraintException`.

[3] Section 7.5.1, "Usage" describes the `enableFailFastCheck` customization and `Section 3.5.2, "Validation"` defines fail-fast checking.

The schema-derived property is annotated, either explicitly or by default mapping annotations, with the mapping annotation @XmlList, specified in Section 8.

6.2.5 Union Property

A union property *prop* is used to bind a union simple type definition schema component. A union simple type definition schema component consists of union members which are schema datatypes. A union property, is therefore, realized by:

```
public Type getId();
public void setId(Type value);
```

where *Id* is a metavariable that represents the Java method identifier computed by applying the name mapping algorithm described in Section D.2, "The Name to Identifier Mapping Algorithm," on page 333 to *prop*.

The *base type* is String. If one of the member types is derived by list, then the Union property is represented as the appropriate collection property as specified by the customization `<jaxb:globalBindings>` **@collectionType** value, specified in Section 7.5.1, "Usage."

- The `getId` method returns the set value. If the property has no set value then the value `null` is returned. The value returned is Type.
- The `setId` method sets the set value.
  If value is `null`, the property's *set value* is discarded. Prior to setting the property's value when TypeConstraint validation is enabled, a non-`null` value is validated by applying the property's predicate, which may throw a `TypeConstraintException`. No setter is generated if the union is represented as a collection property.

Example: Default Binding: Union

The following schema fragment:

```
<xs:complexType name="CTType">
   <xs:attribute name="state" type="ZipOrName"/>
</xs:complexType>
<xs:simpleTypename="ZipOrName"
              memberTypes="xs:integer xs:string"/>
```

is bound to the following Java representation.

```
public class CTType {
    String getState() {...}
    void setState(String value) {...}
}
```

6.2.6 Union

A simple type definition derived by a union is bound using the union property with the following Java property attributes:

- the *base type* as specified in Section 6.2.5, "Union Property."
- if one of the member types is derived by <xs:list>, then the union is bound as a Collection property.
- The *predicate* is the schema constraints specified in "Simple Type Definition Validation Rules," Section 4.1.4 [XSD PART2].

6.3 Complex Type Definition

6.3.1 Aggregation of Java Representation

A Java representation for the entire schema is built based on aggregation. A schema component aggregates the Java representation of all the schema components that it references. This process is done until all the Java representation for the entire schema is built. Hence a general model for aggregation is specified here once and referred to in different parts of the specification.

The model assumes that there is a schema component *SP* which references another schema component *SC*. The Java representation of *SP* needs to aggregate the Java representation of *SC*. There are two possibilities:

- *SC* is bound to a property set.
- *SC* is bound to a Java datatype or a Java value class.

Each of these is described below.

6.3.1.1 Aggregation of Datatype/Class

If a schema component *SC* is bound to a Java datatype or a Java value class, then *SP* aggregates *SC's* Java representation as a simple property defined by:

- **name:** the name is the class/interface name or the Java datatype or a name determined by SP. The name of the property is therefore defined by the schema component which is performing the aggregation.
- **base type:** If SC is bound to a Java datatype, the base type is the Java datatype. If SC is bound to a Java value class, then the base type is the class name, including a dot separated list of class names within which SC is nested.
- **collection type:** There is no collection type.
- **predicate:** There is no predicate.

6.3.1.2 Aggregation of Property Set

If *SC* is bound to a property set, then *SP* aggregates by adding *SC's* property set to its own property set.

Aggregation of property sets can result in name collisions. A name collision can arise if two property names are identical. A binding compiler must generate an error on name collision. Name collisions can be resolved by using customization to change a property name.

6.3.2 Java value class

The binding of a complex type definition to a Java value class is based on the abstract model properties in Section F.1.3, "Complex Type Definition Schema Component," on page 348. The Java value class must be defined as specified here, unless the ref attribute is specified on the `<jaxb:class>` customization, in which case the schema compiler will simply assume that the nominated class is already bound to this complex type.[4]

- **name:** name is the Java identifier obtained by mapping the XML name `{name}` using the name mapping algorithm, specified in Section D.2, "The Name to Identifier Mapping Algorithm," on page 333. For the

[4] Note that Section 6.7.3, "Binding of an anonymous complex type definition" defines the name and package property for anonymous type definitions occurring within an element declaration.

handling of an anonymous complex type definition, see Section 6.7.3, “Binding of an anonymous complex type definition” for how a **name** value is derived from its parent element declaration.

- **package:**
  - For a global complex type definition, the derived Java value class is generated into the Java package that represents the binding of *{target namespace}*
  - For the value of **package** for an anonymous complex type definition, see Section 6.7.3, “Binding of an anonymous complex type definition".
- **outer class name:**
  - There is no outer class name for a global complex type definition.
  - Section 6.7.3, “Binding of an anonymous complex type definition" defines how to derive this property from the element declaration that contains the anonymous complex type definition.
- **base class:** A complex type definition can derive by restriction or extension (i.e. *{derivation method}* is either “extension” or “restriction”). However, since there is no concept in Java programming similar to restriction, both are handled the same. If the *{base type definition}* is itself mapped to a Java value class (Ci2), then the base class must be Ci2. This must be realized as:

```
public class Ci1 extends Ci2 {
    .....
}
```

  See example of derivation by extension at the end of this section.
- **abstract**: The generated Java class is abstract when the complex type definition’s *{abstract}* property is `true`.
- **property set:** The Java representation of each of the following must be aggregated into Java value class’s property set (Section 6.3.1, “Aggregation of Java Representation”).
  - A subset of *{attribute uses}* is constructed. The subset must include the schema attributes corresponding to the `<xs:attribute>` children and the *{attribute uses}* of the schema attribute groups resolved by the <ref> attribute. Every attribute’s Java representation (Section 6.8, “Attribute use”) in the set of attributes computed above must be aggregated.

- If the optional *{attribute wildcard}* is present, either directly or indirectly, a property defined by Section 6.9, “Attribute Wildcard“is generated.
- The Java representation for *{content type}* must be aggregated.

  For a “Complex Type Definition with complex content,” the Java representation for *{content type}* is specified in Section 6.12, “Content Model - Particle, Model Group, Wildcard.”
  For a complex type definition which is a “Simple Type Definition with simple content,” the Java representation for *{content type}* is specified in Section 6.3.2.1, “Simple Content Binding.”
- If a complex type derives by restriction, there is no requirement that Java properties representing the attributes or elements removed by the restriction to be disabled. This is because (as noted earlier), derivation by restriction is handled the same as derivation by extension.

- When the complex type definition’s *{abstract}* property is `false`, a factory method is generated in the package’s `ObjectFactory` class introduced in Section 5.2, “Java Package.” The factory method returns the type of the Java value class. The name of the factory method is generated by concatenating the following components:
  - The string constant `create`.
  - The *name* of the Java value class.

The schema-derived Java value class is annotated, either explicitly or by default mapping annotations, with the mapping annotation @XmlType, specified in Section 8.7.1. The @XmlType annotation elements are derived in terms of the abstract model properties for a complex type definition summarized in Section F.1.3, “Complex Type Definition Schema Component,” on page 348 as follows:

Table 6-1 Annotate Java value class with @XmlType element-value pairs

@XmlType element	@XmlType value
name	complex type definition's {name}
namespace	{target namespace}
propOrder	When {content type} is element-only {content model} and top-level {compositor} is xs:sequence, ordered list of JAXB property names representing order of xs:elements in {content model}. All other cases do not need to set propOrder.

Example: Complex Type: Derivation by Extension

XML Schema Fragment (from XSD PART 0 primer):

```
<xs:complexType name="Address">
    <xs:sequence>
        <xs:element name="name"   type="xs:string"/>
        <xs:element name="street" type="xs:string"/>
        <xs:element name="city"   type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="USAddress">
    <xs:complexContent>
        <xs:extension base="ipo:Address">
            <xs:sequence>
                <xs:element name="state" type="xs:string"/>
                <xs:element name="zip"   type="xs:integer"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Default Java binding:

```
public class Address {
    String getName() {...}
    void   setName(String) {...}
    String getStreet() {...}
    void   setStreet(String) {...}
    void   getCity() {...}
    void   setCity(String) {...}
}

import java.math.BigInteger;
public class USAdress extends Address {
    String getState() {...}
    void   setState(String) {...}{
    BigInteger getZip() {...}
    void   setZip(BigInteger) {...}
}

class ObjectFactory {
    Address createAddress() {...}
    USAddress createUSAddress() {...}
}
```

6.3.2.1 Simple Content Binding

Binding to Property

By default, a complex type definition with simple content is bound to a Java property defined by:

- **name:** The property name must be "`value`".

- **base type, predicate, collection type:** As specified in [XSD Part 1], when a complex type has simple content, the content type (*{content type}*) is always a simple type schema component. And a simple type component always maps to a Java datatype (Section 6.2, "Simple Type Definition"). Values of the following three properties are copied from that Java type:
  - base type
  - predicate
  - collection type

The schema-derived JAXB property representing simple content is annotated, either explicitly or by default mapping annotations, with the mapping annotation @XmlValue, specified in Section 8.9.10.

Example: Simple Content: Binding To Property

XML Schema fragment:

```
<xs:complexType name="internationalPrice">
    <xs:simpleContent>
        <xs:extension base="xs:decimal">
            <xs:attribute name="currency" type="xs:string"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
```

Default Java binding:

```
class InternationalPrice {
    /** Java property for simple content */
    @XmlValue
    java.math.BigDecimal getValue() {...}
    void setValue(java.math.BigDecimal value) {...}

    /** Java property for attribute*/
    String getCurrency() {...}
    void setCurrency(String) {...}
}
```

6.3.3 xsd:anyType

`xsd:anyType` is the root of the type definition hierarchy for a schema. All complex type definitions in a schema implicitly derive from `xsd:anyType`. Given that the JAXB 2.0 architecture does not define a common base class for all JAXB class bindings of complex type definitions, the only possible binding property base type binding for `xsd:anyType` is to `java.lang.Object`. This binding enables all possible type and element substitutions for an element of type `xsd:anyType`.

CODE EXAMPLE 6-1 Binding of element with type `xsd:anyType`

```
<xs:element name="anyContent/> <!--@type defaults to xs:anyType-->
<xs:complexType name="base">
    <xs:sequence>
```

```
        <xs:element ref="anyContent/>
        <xs:element name="anyContentAgain" type="xs:anyType"/>
    </xs:sequence>
</xs:complexType>

public class Base {
    void setAnyContent(Object obj);
    Object getAnyContent();
    void setAnyContentAgain(Object obj);
    Object getAnyContentAgain();
}
```

A schema author defines an element to be of type `xs:anyType` to defer constraining an element to a particular type to the xml document author. Through the use of `xsi:type` attribute or element substitution, an xml document author provides constraints for an element defined as `xs:anyType`. The JAXB unmarshaller is able to unmarshal a schema defined `xsd:anyType` element that has been constrained within the xml document to an easy to access JAXB mapped class. However, when the xml document does not constrain the `xs:anyType` element, JAXB unmarshals the unconstrained content to an element node instance of a supported DOM API.

Type substitution is covered in more detail in Section 6.7.4.1 and 6.7.4.2. Element substitution is covered in more detail in Section 6.7.5.

6.4 Attribute Group Definition

There is no default mapping for an attribute group definition. When an attribute group is referenced, each attribute in the attribute group definition becomes a part of the *[attribute uses]* property of the referencing complex type definition. Each attribute is mapped to a Java property as described in Section 6.8, “Attribute use“. If the attribute group definition contains an attribute wildcard, denoted by the `xs:anyAttribute` element, then the referencing complex type definition will contain a property providing access to wildcard attributes as described in Section 6.9, “Attribute Wildcard“.

6.5 Model Group Definition

When a named model group definition is referenced, the JAXB property set representing its content model is aggregated into the Java value class representing the complex type definition that referenced the named model group definition as illustrated in Section Figure 6.2, “Binding for a reference to a model group definition.”

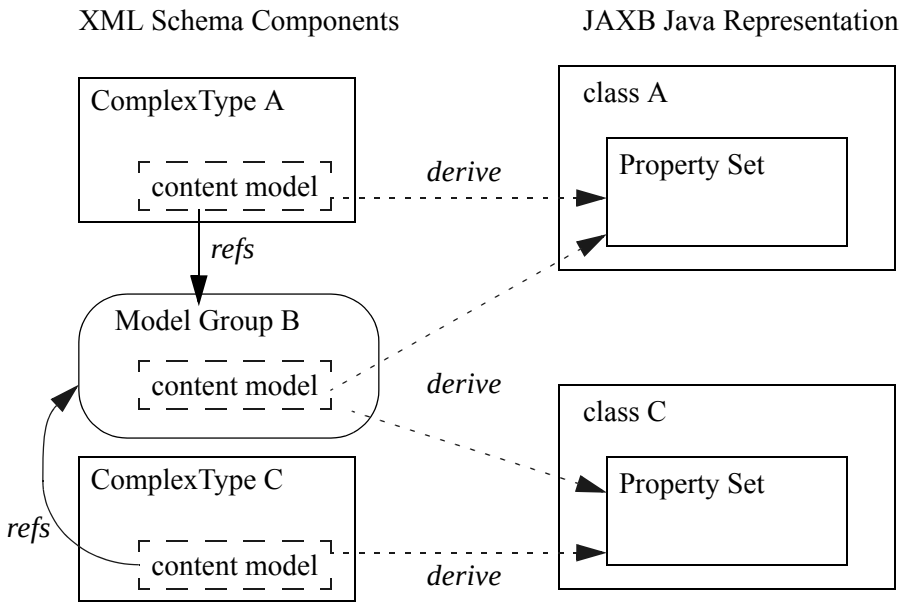


Figure 6.2 Binding for a reference to a model group definition.

This binding style results in the same properties occurring within both Java value class’s A and C to represent the referenced Model Group B’s content model.

When a model group definition’s content model contains an XML Schema component that is to be bound to a Java value class, element class or enum type, it is desirable to only create a single Java representation, not one for each complex content that references the named model group definition. This default binding from a model group definition’s content model is defined in Section 6.5.3, “Deriving Class Names for Named Model Group Descendants."

To meet the JAXB 2.0 goal of predictable unmarshalling of invalid XML content, the JAXB 1.0 customization for binding a model group to a JAXB

mapped class is no longer supported. Section 4.4.4, “Flexible Unmarshalling“details the rationale behind this change.

6.5.1 Bind to a set of properties

A non-repeating reference to a model group definition, when the particle referencing the group has *{max occurs}* equal to one, results in a set of content properties being generated to represent the content model. Section 6.12, “Content Model - Particle, Model Group, Wildcard” describes how a content model is bound to a set of properties and has examples of the binding.

6.5.2 Bind to a list property

A reference to a model group definition from a particle with a repeating occurrence is bound by default as specified in Section 6.12.6, “Bind a repeating occurrence model group“.

Example:

Schema fragment contains a particle that references the model group definition has a *{maxOccurs}* value greater than one.

```
<xs:group name="AModelGroup">
    <xs:choice>
        <xs:element name="A" type="xs:int"/>
        <xs:element name="B" type="xs:float"/>
    </xs:choice>
</xs:group>

<xs:complexType name="foo">
    <xs:sequence>
        <xs:group ref="AModelGroup" maxOccurs="unbounded"/>
        <xs:element name="C" type="xs:float"/>
    </xs:sequence>
</xs:complexType>
```

Derived Java representation:

```
    public class Foo {
    /** A valid general content property of AModelGroup content
        model.*/
    @XmlElements({@XmlElement(type=Integer.class, name="A"),
                  @XmlElement(type=Float.class, name="B")})
    java.util.List<Object> getAModelGroup() {...}

    float getC() {...}
    void setC(float value) {...}
};
```

6.5.3 Deriving Class Names for Named Model Group Descendants

When a model group definition's content model contains XML Schema components that need to be bound to a Java class or interface, this section describes how to derive the package and name for the Java value class, enum type or element class derived from the content model of the model group definition. The binding of XML Schema components to Java classes/interfaces is only performed once when the model group definition is processed, not each time the model group definition is referenced as is done for the property set of the model group definition.

XML Schema components occurring within a model group definition's content model that are specified by this chapter and the customization chapter to be bound to a Java value class, interface or typesafe enum class are bound as specified with the following naming exceptions:

- *package*: The element class, Java value class or typesafe enum class is bound in the Java package that represents the target namespace containing the model group definition.
- *name*: The name of the interface or class is generated as previously specified with one additional step to promote uniqueness between interfaces/classes promoted from a model group definition to be bound to a top-level class within a Java package. By default, a prefix for the interface/class name is computed from the model group definition's *{name}* using the XML name to Java identifier algorithm.If the schema customization **[jaxb:globalBindings]** *@localScoping* has a value of

toplevel, then a prefix is not generated from the model group definition's *{name}*.

For example, given a model group definition named *Foo* containing an element declaration named *bar* with an anonymous complex type definition, the anonymous complex type definition is bound to a Java value class with the name *FooBar*. The following figure illustrates this example.

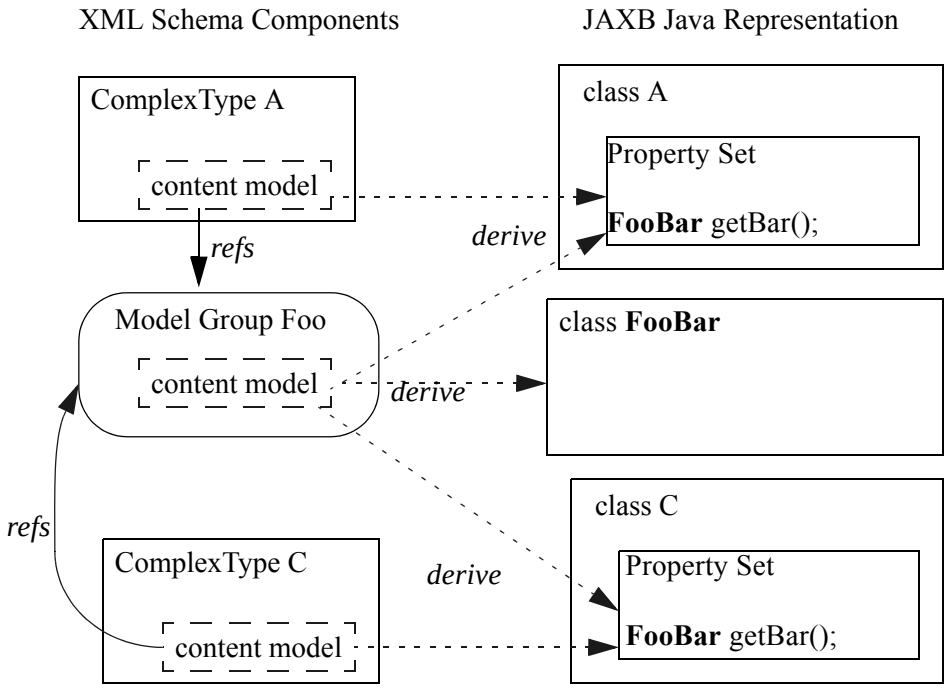


Figure 6.3 Default binding for anonymous type def within a model group definition.

Note that even customization specified Java value class, interface or typesafe enum class names are prepended with the model group definition's name. Thus, if a model group definition named `Foo` contains an anonymous simple type definition with a typesafe enum class customization name of `Colors`, the enum type name is `FooColors`.

6.6 Attribute Declaration

An attribute declaration is bound to a Java property when it is referenced or declared, as described in Section 6.8, “Attribute use,” from a complex type definition.

6.7 Element Declaration

This section describes the binding of an XML element declaration to a Java representation. For a description of how this binding has changed since the previous version, see Section 5.6.3, “Java Element Representation Summary“. This section introduces why a JAXB technology user has to use instances of JAXB element as opposed to instances of Java datatypes or Java value class when manipulating XML content.

An XML element declaration is composed of the following key components:

- its qualified name is *{target namespace}* and *{name}*
- its value is an instance of the Java class binding of its *{type definition}*
- whether the element’s content is *{nillable}*

Typically, an instance of `javax.xml.bind.JAXBElement<T>`, returned by an element factory method, represents an element declaration’s key components. An instance of a Java value class or content interface represents only the value of an element. Commonly in JAXB binding, the Java representation of XML content enables one to manipulate just the value of an XML element, not an actual element instance. The binding compiler statically associates the XML element qualified name to a content property and this information is used at unmarshal/marshal time. For cases where the element name can be dynamically altered at runtime, the JAXB user needs to manipulate elements, not element values. The following schema/derived Java code example illustrates this point.

Example:

Given the XML Schema fragment:

```
<xs:complexType name="chair_kind">
    <xs:sequence>
        <xs:element name="has_arm_rest" type="xs:boolean"/>
    </xs:sequence>
</xs:complexType>
```

Schema-derived Java value class:

```
public class ChairKind {
    boolean isHasArmRest() {...}
    void setHasArmRest(boolean value) {...}
}
```

A user of the Java value class `ChairKind` never has to create a Java instance that both has the value of local element `has_arm_rest` and knows that its XML element name is `has_arm_rest`. The user only provides the value of the element to the content-property `hasArmRest`. A JAXB implementation associates the content-property `hasArmRest` with XML element name `has_arm_rest` when marshalling an instance of `ChairKind`.

The next schema/derived Java code example illustrates when XML element information can not be inferred by the derived Java representation of the XML content. Note that this example relies on binding described in Section 6.12.5, "Bind wildcard schema component."

Example:

```
<xs:complexType name="chair_kind">
    <xs:sequence>
        <xs:any/>
    </xs:sequence>
</xs:complexType>

public class ChairKind {
    @XmlAnyElement(lax="true")
    java.lang.Object getAny() {...}
    void setAny(java.lang.Object elementOrValue) {...}
}
```

For this example, the user can provide an Element instance to the `any` content-property that contains both the value of an XML element and the XML element name since the XML element name could not be statically associated with the content-property `any` when the Java representation was derived from its XML Schema representation. The XML element information is dynamically provided

by the application for this case. Section 6.12, "Content Model - Particle, Model Group, Wildcard," on page 138 cover additional circumstances when one can use JAXB elements.

6.7.1 Bind to JAXBElement<T> Instance

The characteristics of the generated ObjectFactory element factory method that returns an `JAXBElement<T>` instance are derived in terms of the properties of the "Element Declaration Schema Component" on page 349 as follows:

- The element factory method is generated into the `ObjectFactory` class in the Java package that represents the binding of the element declaration's *{target namespace}*.
- The element factory method returns an instance of `javax.xml.bind.JAXBElement<T>`, where `T` is the Java value class representing the *{type definition}* of the element declaration. The factory method sets the element name of the returned instance to the element declaration's fully qualified name.
- The element factory method has a single parameter that is an instance of type `T`, where T is the Java value class representing the *{type definition}* of the element declaration.
- The name of the factory method is generated by concatenating the following components:
  - The string constant `create`.
  - By default, if the element declaration is nested within another XML Schema component, then the concatenation of all outer Java class names representing those XML Schema components. If the schema customization **[jaxb:globalBindings]** *@localScoping* has a value of toplevel, skip this step.
  - A name that is generated from the element declaration's *{name}* using the XML Name to Java identifier name mapping algorithm specified in Section D.2, "The Name to Identifier Mapping Algorithm," on page 333.
- The `JAXBElement<T>` property for nil test whether an element's content model is `xsi:nil="true"`.

For example, an element declaration named `Foo` with a type of `xs:int` that is nested within the content model of complex type definition `Bar` would have the

following factory method generated in the containing Java package's `ObjectFactory` class:

```
JAXBElement<Integer> createBarFoo(Integer value) {...}
```

Default binding rules require an element declaration to be bound to element factory method under the following conditions:

- All non-abstract, named element declarations with global *{scope}* are bound to an element factory method that returns an `JAXBElement<T>` instance. The rationale is that any global element declaration can occur within a wildcard context and one might want to provide element instances, not instances of the element's type, the element's value, for this case.
- All local element declarations, having a *{scope}* of a complex type definition, occurring within content that is mapped to a general content property of JAXB elements must have an element factory method generated. General content property is specified in Section 6.12.3, "General content property" An example of when a content model is mapped to a general content property, forcing the generation of element declarations is at Section 6.12.3.2, "Examples."

The schema-derived element factory method is annotated, either explicitly or by default mapping annotations, with the mapping annotation `@XmlElementDecl`, specified in Section 8. The `@XmlElementDecl` annotation elements are derived in terms of the abstract model properties for an element declaration summarized in Section F.1.4, "Element Declaration Schema Component," on page 349 as follows:

Table 6-1 Annotate element instance factory with @XmlElementDecl element-value pairs.

@XmlElementDecl element	@XmlElementDecl value
name	element declaration's *{name}*
namespace	*{target namespace}*
scope	If *{scope}* is *global*, `JAXBElement.GlobalScope.class`. else the JAXB Java value class representing the *{scope}*ing complex type definition.
substitutionHeadName	If optional *{substitution group affiliation}* exists, its local name.
substitutionHeadNamespace	If optional *{substitution group affiliation}* exists, its namespace.

The element declaration's *{type}* can result in additional JAXB annotations being generated on the element instance factory. For more details, see Section 6.2.2.3, "Annotations for standard XML datatypes"and @XmlList in Section 6.2.4, "List".

The schema-derived ObjectFactory class containing the @XmlElementDecl annotations is annotated with @XmlRegistry annotation.

6.7.2 Bind to Element Class

Section 7.7, "`<class>` Declaration"customization enables the binding of an element declaration with a named type definition to a schema-derived Element class. The characteristics of the schema-derived Element class are derived in

terms of the properties of the "Element Declaration Schema Component" on page 349 as follows:

- The *name* of the generated Java Element class is derived from the element declaration *{name}* using the XML Name to Java identifier mapping algorithm for class names.
- Each generated Element class must extend the Java value class `javax.xml.bind.JAXBElement<T>`. The next bullet specifies the schema-derived Java class name to use for generic parameter `T`.
- If the element declaration's *{type definition}* is
  - Anonymous

    Generic parameter `T` from the second bullet is set to the schema-derived class represented the anonymous type definition generated as specified in Section 6.7.3.
  - Named

    Generic parameter T from the second bullet is set to the Java class representing the element declaration's *{type definition}*.

  The `ObjectFactory` method to create an instance of *name* has a single parameter that is an instance of type `T`. By default, the name of the ObjectFactory method is derived by concatenating *outerClassNames* and *name*. When schema customization, **[jaxb:globalBindings]** *@localScoping,* specified in Section *7.5.1,* has a value of *toplevel*, then the outer Classnames are ommitted from the factory method name.
- If *{scope}* is
  - **Global:** The derived Element class is generated into the Java package that represents the binding of *{target namespace}*.
  - **A Complex Type Definition:** By default, the derived Element class is generated within the Java value class represented by the complex type definition value of *{scope}*. When *@localScoping* is *toplevel*, the derived element class is generated as a toplevel class.
- The property for nil test whether element's content model is `xsi:nil="true"`.
- Optional *{value constraint}* property with pair of `default` or `fixed` and a value.
  If a default or fixed value is specified, the data binding system must

substitute the default or fixed value if an empty tag for the element declaration occurs in the XML content.

A global binding customization, **@generateElementClass,** specified in Section 7.5, "<globalBindings> Declaration"enables this binding over the default binding specified in the previous subsection.

6.7.3 Binding of an anonymous complex type definition

An anonymous complex type definition is bound to a generated schema-derived Java value class by default.

The naming characteristics of the generated Java value class is derived in terms of the properties of the "Element Declaration Schema Component" on page 349 as follows:

- The *name* of the generated Java value class is derived from the element declaration *{name}* using the XML Name to Java identifier
- The *package* of the generated Java value class is the same as the package derived from the element declaration's *{target namespace}*.
- The *outer class names* of the generated Java value class is determined by the element declaration's *{scope}*. If *{scope}* is:
  - Global
    There is no outer class name.
  - A Complex Type Definition
    By default, the derived Java value class is generated nested within the Java value class represented by the complex type definition value of *{scope}*. The derived Java value is not generated nested when schema customization `[globalBindings]` has attribute `@localScoping` with a value of *`toplevel`*.
- *base class:* Same as defined in Section 6.3.2, "Java value class".
- *property set:* As defined in Section 6.3.2, "Java value class".
- A type factory method is generated in the package's `ObjectFactory` class introduced in Section 5.2, "Java Package." The factory method returns the type of the Java value class. The name of the factory method is generated by concatenating the following components:
  - The string constant `create`.

- If the element declaration containing the anonymous complex type definition is nested within another complex type definition representing a value class and [globalBindings] @localScoping has a value of *nested*, then the concatenation of all outer Java class names. This step is skipped when @localScoping has a value of *toplevel*.
- The *name* of the Java value class.

The schema-derived value class is annotated with the mapping annotation `@XmlType`, specified in Section 8.7.1. The `@XmlType` annotation elements are set as described in Table 6-4, "Annotate Java value class with @XmlType element-value pairs," on page 94 with one exception: `@XmlType.name()` is set to the empty string.

As long as the element declaration is not one of the exception cases specified in Section 6.7.3.1, "Bind Element Declaration to JAXBElement", the schema-derived value class is annotated with the mapping annotation `@XmlRootElement` specified in Section 8. The `@XmlRootElement` annotation elements are derived in terms of the abstract model properties for the referenced global element declaration summarized in Section F.1.4, "Element Declaration Schema Component," on page 349 as follows:

Table 6-2 Annotate JAXB Mapped Class with @XmlRootElement element-value pairs

@XmlRootElement element	@XmlRootElement value
namespace	When element declaration *{target namespace}* is absent, (i.e. unqualified local element declaration), @XmlElement.namespace() is not set. Otherwise, set @XmlElement.namespace() to value of *{target namespace}*. (either a qualified local element declaration or a reference to a global element) Note: same result could be achieved with package level annotation of @XmlSchema and not setting @XmlElement.namespace.
name	element declaration *{name}*

Example:

Given XML Schema fragment:

```
<xs:element name="foo">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="bar" type="xs:int"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Derived Java code:

```
/*Value class representing element declaration with an anonymous
complex type definition.*/
@XmlType(name="")
@XmlRootElement(namespace="", name="foo")
public class Foo {
    int getBar() {...}
    void setBar(int value) {...}
};
```

6.7.3.1 Bind Element Declaration to JAXBElement

An element declaration with an anonymous complex type definition is not bound to a `@XmlRootElement`,annotated schema-derived class when the element declaration is:

- nillable
- the head element or a member of a substitution group
- non-global (i.e. declared within a complex type definition)

When one or more of the above conditions are met, the schema-derived class representing the anonymous complex type definition must not be annotated with `@XmlRootElement`. Instead, an element factory that returns `JAXBElement<`*`anonymousTypeValueClass`*`>` may be generated as specified in Section 6.7.1, “Bind to `JAXBElement<T>` Instance“.

Example:

Given XML Schema fragment:

```
<xs:element name="foo" nillable="true">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="bar" type="xs:int"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Derived Java code:

```
/*Value class representing anonymous complex type definition.*/
@XmlType(name="")
public class Foo {
    int getBar() {...}
    void setBar(int value) {...}
};
@XmlRegistry
class ObjectFactory {
    // type factory method
    Foo               createFoo() {...}
    // element factory method
    @XmlElementDecl(name="foo", namespace="", nillable="true")
    JAXBElement<Foo> createFoo(Foo value) {...}
}
```

6.7.4 Bind to a Property

A local element declaration is bound by default to a Java property as described in Section 5.5, “Properties,” on page 54. The characteristics of the Java property are derived in terms of the properties of the “Element Declaration Schema Component” on page 349 and “Particle Schema Component” on page 351 as follows:

- The *name* of the Java property is derived from the *{element declaration}* property’s *{name}* property using the XML Name to Java Identifier mapping algorithm described in Section D.2, “The Name to Identifier Mapping Algorithm,” on page 333.
- A *base type* for the Java property is derived from the `{element declaration}` property’s `{type definition}` property as described in binding of Simple Type Definition in Section 6.2, “Simple

Type Definition." or Section 6.3, "Complex Type Definition". If the base type is initially a primitive type and this JAXB property is *optional*, the **[jaxb:globalBinding]** customization `@optionalProperty` controls the binding of an optional primitive property as described in Section 7.5.1, "Usage".

- An optional *predicate* for the Java property is constructed from the `{element declaration}` property's `{type definition}` property as described in the binding of simple type definition to a Java representation.
- An optional *collection type* for the Java property is derived from:
  - `{element declaration}` property's `{type definition}` property as described in the binding of simple type definition to a Java representation
  - the {particle} property's {max occurs} value being greater than one.
- Element defaulting
  The default value is derived from the element declaration's {value constraint} property's value. Unlike attribute defaulting, an element only defaults when there is an empty element tag in an xml document. The element's default value is captured by mapping annotation `@XmlElement.defaultValue()`. The unmarshaller sets the property to this default value when it encounters an empty element tag. The marshaller can output an empty element tag whenever the element's `@XmlValue` property value is the same as its defaulted value..
- A local element declaration that binds to a JAXB property with a primitive base type is bound as an *optional* JAXB property if the element declaration is a member of a choice model group or the element declaration's particle has optional occurrence, `{min occurs} value is "0",` or belongs to a model group that has optional occurrence. By default, the optional JAXB property binds the property's base type to the Java wrapper class for the primitive type. One can test and set the absence of an optional property using null. The **[jaxb:globalBinding]** customization `@optionalProperty` controls alternative bindings of an optional primitive property as described in Section 7.5.1, "Usage".
- If the element declaration's *{nillable}* property is `"true"`, the base type for the Java property is mapped to the corresponding Java wrapper class for the Java primitive type. Setting the property to the `null` value indicates that the property has been set to the XML Schema concept of @xs:`nil='true'`.

This Java property is a member of the Java value class that represents the binding of the complex type definition containing the local element declaration or reference to global element.

The schema-derived JAXB property getter method is annotated, either explicitly or by default mapping annotations, with the mapping annotation `@XmlElement`, specified in Section 8, "@XmlElement". The `@XmlElement` annotation elements are derived in terms of the abstract model properties for the referenced global element declaration summarized in Section F.1.4, "Element Declaration Schema Component," on page 349 as follows:

Table 6-1 Annotate JAXB Property with @XmlElement element-value pairs

@XmlElement element	@XmlElement value
namespace	When element declaration *{target namespace}* is absent, (i.e. unqualified local element declaration), @XmlElement.namespace() is not set. Otherwise, set @XmlElement.namespace() to value of *{target namespace}*. (either a qualified local element declaration or a reference to a global element) Note: same result could be achieved with package level annotation of @XmlSchema and not setting @XmlElement.namespace.
name	element declaration *{name}*
nillable	element declaration *{nillable}*
defaultValue	if element declaration *{value constraint}* is not absent, set defaultValue() to *{value constraint}*'s value.

CODE EXAMPLE 6-14illustrates how to define an element substitution group and to reference the head element of the substitution group within an Xml Schema. CODE EXAMPLE 6-15 illustrates the Java bindings of the element substation enabled schema. CODE EXAMPLE 6-16 demonstrates element substitution using the JAXB API.CODE EXAMPLE 6-17 illustrates invalid element substitution handling.

6.7.4.1 Type Substitution of a Complex Type Definition

Section 6.3, "Complex Type Definition"describes that when a complex type definition is mapped to Java value class that the type definition derivation

hierarchy is preserved in the Java class hierarchy. This preservation makes it quite natural for Java to support the Xml Schema mechanism type substitution across all complex type definitions.

Performing an invalid type substitution is not detected as a fail-fast check when setting the JAXB property or checked as part of marshalling the element declaration. Invalid type substitution can be checked by optional validation that can be enabled as part of unmarshalling or marshalling process.

The following three code examples illustrate how type substitution is supported in JAXB 2.0 for a complex type definition hierarchy.

CODE EXAMPLE 6-1 Xml Schema example containing type derivation hierarchy

```
<xs:schema targetNamespace="travel:acme" xmlns:a="travel:acme">

<!-- Define type definition derivation hierarchy -->
<xs:complexType name="TransportType">...<\xs:complexType>
<xs:complexType name="PlaneType">
   <xs:extension base="a:TransportType">...<\xs:complexType>
<xs:complexType name="AutoType">
   <xs:extension base="a:TransportType">...<\xs:complexType>
<<xs:complexType name="SUV">
   <xs:extension base="a:AutoType">...<\xs:complexType>

<xs:complexType name="itinerary">
   <xs:sequence>
      <!-- Type substitution possible for "transport".-->.
      <xs:element name="transport" type="TransportType"/>
   </xs:sequence>
</xs:complexType>
</xs:schema>
```

CODE EXAMPLE 6-2 Java binding of Xml Schema from CODE EXAMPLE 6-14

```
package travel.acme;

// Type derivation hierarchy from schema is preserved in Java binding.
public class TransportType { ... }
public class PlaneType extends TransportType { ... }
public class AutoType extends TransportType { ... }
public class SUV extends AutoType { ... }

public class ObjectFactory {
   // Type Factories
   TransportType createTransportType(){...}
   AutoType createAutoType(){...}
   PlaneType createPlaneType(){...}
   TrainType createSUV(){...}
}

public class Itinerary {
   // Simple property supports type substitution.
   TransportType getTransport(){...}
   void setTransport(TransportType value)
}
```

CODE EXAMPLE 6-3 Type substitution using Java bindings from CODE EXAMPLE 6-15

```
ObjectFactory of = ...;
Itinerary itinerary = of.createItinerary();
itinerary.setTransport(of.createTransportType); // Typical Use

// Type Substitution
// transport marshalled as <e:transport xsi:type="e:AutoType">>
itinerary.setTransport(of.createAutoType());

// transport marshalled as <e:transport xsi:type="e:PlaneType">>
itinerary.setTransport(of.createPlaneType());
```

6.7.4.2 Type Substitution of a Simple Type Definition

An XML element declaration having a simple type definition is bound most naturally to a JAXB property with a base type that is a primitive Java datatype. Unfortunately, this strongly typed binding conflicts with fully supporting type substitution of a simple type definition. Unlike the JAXB binding of complex type definitions, the simple type derivation hierarchy is not preserved when binding builtin XML Schema simple type definitions to corresponding Java

datatypes as specified in Section 6.2.2, “Atomic Datatype“. Since there is not a natural Java inheritance hierarchy to support simple type substitution, a JAXB property customization is required to enable optimal support of simple type substitution.

For example, the most natural binding of an XML Schema built-in datatype `xs:int` is to the Java primitive datatype, `int`. However, simple type substitution implies that an `xs:short` or a complex type definition that derives by extension from `xs:int` can be type substituted for an `xs:int` within an XML document using the `xsi:type` attribute. The strongly typed JAXB property with Java type `int` would never allow for a Java value class for the complex type to be assigned to a JAXB property of type *int*.

By default, unmarshalling handles simple type substitution by assigning the relevant part of the type substituted content to the JAXB property. When the value of the xsi:type attribute resolves to:

- a type that derives by restriction from the element’s schema type. The substituted value is always parsable into a legal value of the base type of the JAXB property being type substituted.
- a complex type that derives by extension from element’s schema type. The JAXB binding of the substituted complex type definition must have one JAXB property annotated with an `@XmlValue` that is assignable to the type substituted JAXB property’s base type. Attribute(s) associated with the complex type definition can not be preserved by the default binding.

The rationale behind the default binding is that substitution of a simple type definition occurs rarely. The default JAXB binding is more convenient and precise for programmer to use. Its one drawback is that it does not faithfully preserve `xsi:type` occurring in an XML document.

To enable more comprehensive support of simple type substituting of an XML element with a simple type definition, the JAXB property customization specified in Section 7.8.2.2, “Generalize/Specialize baseType with attribute @name“enables setting the property’s base type to the more general type of `java.lang.Object`. This binding allows for retention of the XML document `xsi:type` and attributes associated with complex type definition substituted for an XML element with a simple type definition. When an `xsi:type` value refers to a type definition not registered with `JAXBContext` instance, the content is unmarshalled as the element’s schema type.

To preserve an application-defined simple type definition involved in simple type substitution, it must be mapped to a JAXB mapped class as described in Section 6.2.2.2, “Bind to a JAXB mapped class“. This can be achieved for all simple type definitions in a schema using the customization `<jaxb:globalBinding mapSimpleTypeDefs="true"/>` or it can be achieved per simple type definition using <jaxb:class> customization. An invalid simple type substitution can be detected by JAXP 1.3 validation enabled at unmarshal or marshal time

Below are examples of the type substitution of an XML element’s simple type definition for the default and customized binding.

CODE EXAMPLE 6-1 Schema fragment to illustrate simple type substitution

```
<xsd:element name="Price">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <!-- element price subject to type substitution->
            <xsd:element name="price" type="xsd:int"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:complexType name="AmountType">
    <xsd:simpleContent> <!-- type substitutable for xs:int -->
        <xsd:extension base="xsd:int">
        <xsd:attribute name="currency" type="xsd:string"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
<xsd:simpleType name="AppInt">
    <xsd:restriction base="xsd:int"/>
</xsd:simpleType>
```

CODE EXAMPLE 6-2 XML documents with simple type substitution

```
<product>
    <name>hotdog</name><price>3</price>
</product>
<product>
    <name>peanuts</name>
    <price xsi:type="short">4</price>
</product>
<product>
    <name>popcorn</name>
    <price xsi:type="AppInt">5</price>
</product>
<product>
    <name>sushi</name>
    <price xsi:type="AmountType" currency="yen">500</price>
</product>
```

Default Handling of Simple Type Substitution

CODE EXAMPLE 6-3 Default JAXB binding of CODE EXAMPLE 6-10

```
public class AmountType {
    @XmlValue
    int getValue(){...} void setValue(int value) {...}
    String getCurrency() {...} void setCurrency(String value) {...}
}
@XmlRootElement(namespace="", name="product")
public class Product {
    int getPrice(){...} void setPrice(int value) { ...}
    int getName() {...} void setName(String value) {...}
}
```

Unmarshalling XML document fragments from CODE EXAMPLE 6-11 into CODE EXAMPLE 6-12 JAXB binding of element `product` results in the `xsi:type` and attributes associated with JAXB mapped class `Price` being lost as part of the unmarshal process. This loss is illustrated by comparing Table 6-8 with Table 6-9.

Table 6-1 Product instances from unmarshalling XML docs from CODE EXAMPLE 6-11

document xsi:type	Product .name value	Product .price value	Product .price type	marshal Product. price xsi:type
	hotdog	3	int	
xs:short	peanuts	4	int	
AppInt	popcorn	5	int	
AmountType	sushi	500	int	

Simple Type Substitution enabled by JAXB customizations.

The simple type definition `AppInt` is mapped to a JAXB class either by `<jaxb:class>` customization or by `<jaxb:globalBindings mapSimpleTypeDef="true"/>`. The JAXB property `Product.Price` is mapped to a JAXB property with a general base type of `java.lang.Object` with following external JAXB schema customization:

```
<jaxb:bindings schemaLocation="CODE EXAMPLE 6-10"
               node="//xsd:element[@name='price']">
   <jaxb:property>
      <jaxb:baseType name="java.lang.Object"/>
   </jaxb:property>
</jaxb:bindings>
```

specified in Section 7.8.2.2, “Generalize/Specialize baseType with attribute @name“.

CODE EXAMPLE 6-4 Customized JAXB binding of CODE EXAMPLE 6-10

```
public class AmountType {
    @XmlValue
    int getValue(){...} void setValue(int value) {...}
    String getCurrency() {...} void setCurrency(String value) {...}
}
public class AppInt {
    @XmlValue
    int getValue() {...} void setValue(int value) {...}
}
public class Product {
    // enable simple type substitution with base type of Object
    @XmlElement(type=java.lang.Integer.class)
    Object getPrice(){...} void setPrice(Object value) { ...}
    int getName() {...} void setName(String value) {...}
}
```

Unmarshalling XML document fragments from CODE EXAMPLE 6-11 into CODE EXAMPLE 6-13 JAXB binding of element `product` preserves the `xsi:type` and attributes associated with JAXB mapped class `AmountType` is illustrated in Table 6-9.

Table 6-2 Product instances from unmarshalling XML docs from CODE EXAMPLE 6-11

document xsi:type	Product .name value	Product. price value	Product. price Java type	Marshal Product. price xsi:type
	hotdog	3	Integer	
xs:short	peanuts	4	Short	xs:short
AppInt	popcorn	5	AppInt	AppInt
AmountType	sushi	{value=500, currency="yen"}	AmountType	AmountType

6.7.5 Bind to a Simple Element property

Element substitution group is an Xml Schema mechanism that enables the substitution of one named element for another. This section uses terms and

concepts described in Section 4.6 of [XSD Part 0] and normatively defined in Section 2.2.2.2 of [XSD Part 1].

The following constraints assist in defining the Java binding that enables element substitution group:

1. Element substitution is only possible for a reference to a global element.
    a. Assuming the absence of the Xml Schema constraints on substitution, any global element can be made the head element of a substitution group.
2. **All elements in a substitution group must derive from or have the same type definition as the head element.**

To support element substitution, for each global element reference to a head element of a substitution group or to an abstract element, it is necessary to generate the Element property bindings defined in Section 5.5.5, "Element Property," on page 63.[5] This property enables the overriding of the schema-specified element name bound to a JAXB property by setting and getting the JAXB element representation, `javax.xml.bind.JAXBElement<T>`. The name property of the JAXBElement<T> instance overrides the schema specified element declaration name. To enable the passing of any element that could be part of the element substitution group, it is necessary to accept any JAXBElement derivation that extends Java binding of the head element's type definition. Using the upper bounded wildcard notation for a generic JAXBElement container, `JAXBElement<? extends T>`, the element property is able to get and set any element that has an element value that is a subtype of T. Compile time checking will not allow invalid JAXBElement derivations to be passed to the Element property setter. When the element type is correct but the element name is not part of the substitution group, this invalid scenario can only be caught at runtime by validation or optional fail-fast checking by the element property setter.[6]

[5] Element substitution extensibility does allow element substitution(s) to be defined in a separate schema than a global element reference occurs. When schemas are not compiled at same time, the schema to java binding declaration, <jaxb:property generateElementProperty="true"/> described in 7.8.1 forces the generation of an element property for a global element reference, independent of it not belonging to a element substitution group.

The schema-derived Element property getter method is annotated, either explicitly or by default mapping annotations, with the mapping annotation `@XmlElementRef`, specified in Section 8.10.3, "@XmlElementRef". The `@XmlElementRef` annotation elements are derived in terms of the abstract model properties for the referenced global element declaration summarized in Section F.1.4, "Element Declaration Schema Component," on page 349 as follows:

Table 6-3 Annotate Element Property with @XmlElementRef element-value pairs

@XmlElementRef element	@XmlElementRef value
value	javax.xml.bind.JAXBElement.class
namespace	referenced element declaration *{target namespace}*
name	referenced element declaration *{name}*

CODE EXAMPLE 6-14illustrates how to define an element substitution group and to reference the head element of the substitution group within an Xml Schema. CODE EXAMPLE 6-15 illustrates the Java bindings of the element substation enabled schema. CODE EXAMPLE 6-16 demonstrates element substitution using the JAXB API.CODE EXAMPLE 6-17 illustrates invalid element substitution handling.

6. The desire to reduce the overall number of schema-derived classes generated by default influenced the decision to default to binding an element declaration to an element instance factory. A customization described in Section 7.5, "`<globalBindings>` Declaration exists that binds each element declaration to a Java element class so element substitution checking can be enforced entirely by strongly typed method signatures.

CODE EXAMPLE 6-5 Xml Schema example containing an element substitution group

```
<xs:schema targetNamespace="travel:acme" xmlns:a="travel:acme">

<!-- See type definition derivation hierarchy defined in CODE EXAMPLE 6-7for
    complexType definitions TransportType, PlaneType, AutoType and SUV.-->
<
<!-- Define element substitution group. a:transport is head element.-->
<xs:element name="transport" type="a:TransportType"/>
<xs:element name="plane" type="a:PlaneType" substitutionGroup="a:transport"/>
<xs:element name="auto" type="a:AutoType" substitutionGroup="a:transport"/><!-

<xs:complexType name="itinerary">
    <xs:sequence>
        <!-- Global element reference.
             References head element of element substitution group. -->
        <xs:element ref="a:transport"/>
    </xs:sequence>
</xs:complexType>
<\xs:schema>
```

CODE EXAMPLE 6-6 avo binding of Xml Schema from CODE EXAMPLE 6-14

```
ppackage travel.acme;
public class ObjectFactory {
   // Type Factories
   TransportType createTransportType();
   AutoType createAutoType();
   PlaneType createPlaneType();
   TrainType createSUVType();

   // Element Instance Factories
   JAXBElement<AutoType> createAuto(AutoType value);
   JAXBElement<PlaneType> createPlane(PlaneType value);
   JAXBElement<TransportType> createTrain(TransportType value);
}

// See Java binding of type derivation hierarchy in CODE EXAMPLE 6-8

public class Itinerary {
   // Element substitution supported by Section 5.5.5, "Element Property"
   JAXBElement<? extends TransportType> getTransport();
   void setTransport(JAXBElement<? extends TransportType> value);
}
```

CODE EXAMPLE 6-7 Element substitution using Java bindings from CODE EXAMPLE 6-15

```
ObjectFactory of = ...;
Itinerary itinerary = of.createItinerary();
itinerary.setTransport(of.createTransportType()); // Typical use.

// Element substitution:
// Substitute <e:auto> for schema specified <e:transport>.
itinerary.setTransport(of.createAuto(of.createAutoType()));

// Substitute <e:plane> for schema specified <e:transport>>
itinerary.setTransport(of.createPlane(of.createPlaneType()));

// Combination of element and type substitution:
// Substitutes <e:auto xsi:type="e:SUV"> for <e:transport>>
itinerary.setTransport(of.createAuto(of.createSUV()));
```

CODE EXAMPLE 6-8 Invalid element substitution using Java bindings from CODE EXAMPLE 6-15

```
<!-- Add elements not part of element substitution group. -->
<xs:element name="apple" type="xsd:string"/>
   <xs:complexType name="spaceShuttle">
   <xs:extension base="a:TransportType">...<\xs:complexType>
<xs:element name="spaceShuttle" type="a:spaceShuttleType">

ObjectFactory of = ...;
Itinerary itinerary = of.createItinerary();
// Invalid element substitution
// compile time error: method not found
//Element apple of type JAXBElement<String> does not match
//bounded wildcard JAXBElement<? extends TransportType>.
itinerary.setTransport(of.createApple("granny smith"));

//Invalid element substitution detected by validation.
// Element spaceShuttle not part of substitution group.
// Adding substitutionGroup="transport" to line 4 fixes this.
itinerary.setTranport(
      of.createSpaceShuttle(of.createSpaceShuttleType()));
```

6.7.6 Bind to an Element Collection property

A repeating occurrence element declaration that is element substitutable binds solely to a JAXB Collection property of JAXBElement.

CODE EXAMPLE 6-9 Bind repeating occurrence element substitution variant of CODE EXAMPLE 6-14

```
<!--deleted schema that remains same -->
<xs:complexType name="itinerary">
    <xs:sequence>
    <!-- Repeating occurance to substitutable global element reference. -->
        <xs:element ref="a:transport" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<\xs:schema>

Java Binding:
public class Itinerary {
    List<JAXBElement<? extends TransportType>> getTransport();
}
```

6.8 Attribute use

A ‘required’ or ‘optional’ attribute use is bound by default to a Java property as described in Section 5.5, “Properties,” on page 54. The characteristics of the Java property are derived in terms of the properties of the “Attribute Use Schema Component” on page 351 and “Attribute Declaration Schema Component” on page 350 as follows:

- The *name* of the Java property is derived from the *{attribute declaration}* property’s *{name}* property using the XML Name to Java Identifier mapping algorithm described in Section D.2, “The Name to Identifier Mapping Algorithm,” on page 333.
- A *base type* for the Java property is derived from the `{attribute declaration}` property’s `{type definition}` property as described in binding of Simple Type Definition in Section 6.2, “Simple Type Definition.”. If the base type is initially a primitive type and this JAXB property is *optional*, the **`[jaxb:globalBinding]`** customization `@optionalProperty` controls the binding of an optional primitive property as described in Section 7.5.1, “Usage“.

- An optional *predicate* for the Java property is constructed from the `{attribute declaration}` property's `{type definition}` property as described in the binding of simple type definition to a Java representation.
- An optional *collection type* for the Java property is derived from the `{attribute declaration}` property's `{type definition}` property as described in the binding of simple type definition to a Java representation.
- The *default value* for the Java property is the *value* from the attribute use's *{value constraint}* property. If the optional *{value constraint}* is absent, the default value for the Java property is the Java default value for the base type.
- The JAXB property is *optional* when the attribute use's `{required}` property is `false`.

This Java property is a member of the Java value class that represents the binding of the complex type definition containing the attribute use

The JAXB property getter for this attribute is annotated, either explicitly or via default mapping, with the mapping annotation @XmlAttribute, specified in Section 8.9.7. The @XmlAttribute element values are derived in terms of the properties of the "Attribute Use Schema Component" on page 351 and "Attribute Declaration Schema Component" on page 350 as follows:

Table 6-4 Annotate Attribute property getter method with @XmlAttribute annotation

@XmlAttribute element	@XmlAttribute value
name	attribute declaration's {name}
namespace	if attribute declaration's {target namespace} absent, set to "" otherwise, set to {target namespace}
required	attribute use's {required}

Design Note – Since the target namespace is not being considered when mapping an attribute to a Java property, two distinct attributes that have the same *{name}* property but not the same *{target namespace}* will result in a Java property naming collision. As specified generically in Section D.2.1, "Collisions and conflicts," on page 336, the binding compiler detect this name collision between the two distinct properties and reports the error. The user can provide a customization that provides an alternative Java property name to resolve this situation.

Example:

Given XML Schema fragment:

```
<xs:complexType name="USAddress">
    <xs:attribute name="country" type="xs:string"/>
</xs:complexType>
```

Default derived Java code:

```
public class USAddress {
    @XmlAttribute(name="country", targetNamespace="",
                  required="false");
    public String getCountry() {...}
    public void setCountry(String value) {...}
}
```

6.8.1 Bind to a Java Constant property

Rather than binding to a read/write JAXB property, an attribute use with a `fixed` *{value constraint}* property can be bound to a Java Constant property. This mapping is not performed by default since `fixed` is only a validation constraint. The user must set the binding declaration attribute `fixedAttributeToConstantProperty` on `<jaxb:globalBinding>` element as specified in Section 7.5.1, "Usage," on page 166 or on `<jaxb:property>` element as specified in Section 7.8.1, "Usage," on page 182 to enable this mapping.

Example:

Given XML Schema fragment:

```
<xs:annotation><xs:appinfo>
```

```
    <jaxb:globalBindings fixedAttributeAsConstantProperty="true"/>
</xs:appinfo></xs:annotation>
<xs:complexType name="USAddress">
    <xs:attribute name="country" type="xs:NMTOKEN" fixed="US"/>
</xs:complexType>
```

If the appropriate binding schema customization enables mapping a fixed XML value to Java constant property, the following Java code fragment is generated.

```
public class USAddress {
    @XmlAttribute
    public static final String COUNTRY="US";
    ...
}
```

The schema-derived constant for this fixed attribute is annotated, either explicitly or via default mapping, with the mapping annotation `@XmlAttribute`. The elements of `@XmlAttribute` are set as described in Table 6-11

Note that if derivation by restriction constrains an existing attribute declaration to be fixed, this refinement must not be bound to a constant property. The initial binding of the attribute to a JAXB property remains the only binding of the attribute in the Java class hierarchy.

6.8.1.1 Contributions to Local Structural Constraint

If the attribute use's *{required}* property is true, the local structural constraint for an instance of the Java value class requires that the corresponding Java property to be set when the Java value class instance is validated.

6.8.2 Binding an IDREF component to a Java property

An element or attribute with a type of `xs:IDREF` refers to the element in the instance document that has an attribute with a type of `xs:ID` or derived from type `xs:ID` with the same value as the `xs:IDREF` value. Rather than expose the Java programmer to this XML Schema concept, the default binding of an `xs:IDREF` component maps it to a Java property with a base type of java.lang.Object. The caller of the property setter method must be sure that its parameter is identifiable. An object is considered identifiable if one of its properties is derived from an element or attribute that is or derives from type

`xs:ID`. The JAXB mapped class must have one property annotated with an @XmlID program annotation as it is specified in Section 8. There is an expectation that all instances provided as values for properties' representing an `xs:IDREF` should have the Java property representing the `xs:ID` of the instances set before the content tree containing both the `xs:ID` and `xs:IDREF` is marshalled. If a property representing an `xs:IDREF` is set with an object that does not have its `xs:ID` set, the `NotIdentifiableEvent` is reported by marshalling.

- The *name* of the Java property is derived from the *{name}* property *of the attribute or element* using the XML Name to Java Identifier mapping algorithm described in Section D.2, "The Name to Identifier Mapping Algorithm," on page 333.
- A *base type* for the Java property is java.lang.Object.
- There is no *predicate* for a property representing an `xs:IDREF`.
- An optional *collection type*
- Default and fixed values can not be supported for an attribute with type `xs:IDREF`.

The schema-derived JAXB property representing xs:IDREF(s) is annotated, either explicitly or by default mapping annotations, with the mapping annotation @XmlIDREF, specified in Section 8.

Example:

Given XML Schema fragment:

```
<xs:complexType name="Book">
    <xs:sequence>
        <xs:element name="author" type="xs:IDREF"/>
        <!-- ... -->
    </xs:sequence>
</xs:complexType>
<xs:complexType name="AuthorBio">
    <xs:sequence><!-- ... --> </xs:sequence>
    <xs:attribute name="name" type="xs:ID"/>
</xs:complexType>
```

Schema-derived Java value class:

```
public class Book {
    @XmlIDREF
    java.lang.Object getAuthor() {...}

    /** Parameter referencedObj should have an attribute or
     * child element with base type of xs:ID by validation
     * or marshal time.
     */
    void setAuthor(java.lang.Object referencedObj){...}
}
public class AuthorBio{
    @XmlID
    String getName(){...}
    void setName(String value){...}
}
```

Demonstration of a Java content instance referencing another instance:

```
Book book = ...;
AuthorBio authorBio = ...;
book.setAuthor(authorBio);
authorBio.setName("<some author's name>");
// The content instance root used to validate or marshal book must
// also include "authorBio" as a child element somewhere.
// A Java content instance is not included
```

Note that ID and IDREF mechanisms do not incorporate type definitions that can be referenced. To generate stronger typing for a JAXB property representing an IDREF, the schema customization described in Section 7.8.2.2, "Generalize/ Specialize baseType with attribute @name" can be used to specialize the binding.. CODE EXAMPLE 7-4 illustrates the generation of stronger typing for the above example.

6.9 Attribute Wildcard

Attribute wildcard is an extensibility feature in XML Schema. It enables an XML document author to introduce attribute(s) to an element that were not statically associated with the element's complex type definition. Obviously, it is not possible to bind such an attribute to a strongly typed JAXB property as the previous section describes for attribute use schema component. The JAXB binding of a complex type definition that contains an attribute wildcard, directly or indirectly, provides dynamic access to the wildcard attributes via the following property:

```
// Return, by reference, a mapping of attribute QName and String.
Map<QName, String> getOtherAttributes();
```

The returned attribute map provides dynamic access to wildcard attributes associated with a complex type definition. The key to the map is the attribute's QName and the key's value is the String value of the attribute.

The schema-derived property getter method is annotated, either explicitly or by default mapping annotations, with the mapping annotation `@XmlAnyAttribute`, specified in Section 8.

The following code examples show the JAXB binding for xs:anyAttribute and how to manipulate wildcard attributes using this binding.

CODE EXAMPLE 6-1 Bind anyAttribute to a JAXB property

```
<xs:schema targetNamespace="http://a.org">
   <xs:complexType name="widget">
       <xs:anyAttribute/>
       <xs:attribute name="color" type="xs:string"/>
   </xs:complexType>
</xs:schema>

package org.a;
importjavax.xml.namespace.QName;
import java.util.Map;
public class Widget {
   String  getColor(){...}
   void    setColor(String value){...}
   @XmlAnyAttribute Map<QName, String> getOtherAttributes(){...}
}
```

CODE EXAMPLE 6-2 Dynamic access to wildcard attribute and attribute use

```
import javax.xml.bind.DatatypeConverter;
Widget w = ...;
Map attrs = w.getOtherAttributes();

// access schema-defined global attribute associated with
// complexType defintion widget via attribute wildcard.
QName IS_OPEN=new QName("http://example.org", "isOpen");
boolean isOpen =
DatatypeConverter.parseBoolean(attrs.get(IS_OPEN));

// set wildcard attribute value
attrs.put(IS_OPEN, DatatypeConverter.printBoolean(false));

// semantically the same results setting attribute use via
// dynamic or static setter for attribute use.
attrs.put(new QName("color"), "red");

// iterate over wildcard attributes
for (Map.Entry<QName,String> e: attrs.entrySet()) {
      System.out.println("Attribute: " + e.getKey() +
                         " Value:" + e.getValue());
}
```

6.10 Redefine

Redefinition allows an existing XML Schema component to be “renamed” and its new definition takes the place of the old one. The binding of the redefined schema components, simple and complex type definitions and model and attribute group declarations, are described in the following subsections.

6.10.1 Bind Redefined Simple Type Definition

As introduced in Section 6.2, “Simple Type Definition“, a schema component using a simple type definition typically binds to a JAXB property. The base type, collection type and predicate of the JAXB property are derived from the simple type definition. Thus, the redefine of a simple type definition results in

the redefinition of the simple type definition being used to derive the base type, collection type and predicate of the JAXB property.

The one exception to this binding is that a simple type definition with enum facets is sometimes bound to an enum type. A redefined simple type definition that binds to an enum type, as described in Section 6.2.3, “Enum Type“, is not bound to a Java representation, only the redefinition is bound to an enum type.

6.10.2 Bind Redefined Complex Type Definition

A redefinition of a type definition must use the original type definition as its base type definition. The redefined complex type definition is bound to a Java value class or interface name that prepends “_” to the class name. The redefinition complex type definition is bound to a class that extends the JAXB class that represents the redefined complex type definition.

CODE EXAMPLE 6-3 Binding of a redefined complex type definition

```
File: v1.xsd:
<!-- Extracted from Section 4.2.2 of [XSD1] -->
<xs:complexType name="personName">
    <xs:sequence>
        <xs:element name="title" type="xs:string" minOccurs="0"/>
        <xs:element name="forename" type="xs:string"
                  minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
File: v2.xsd:
<xs:redefine schemaLocation="v1.xsd">
    <xs:complexType name="personName">
        <xs:complexContent>
            <xs:extension base="personName">
                <xs:sequence>
                    <xs:element name="generation" minOccurs="0"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:redefine>
```

```
Java binding:
// binding of file v1.xsd complex type definition for personName
@XmlType(name="_PersonName")
public class _PersonName {
    void setTitle(String value); String getTitle();
    List<String> getForename();
}
// binding of v2.xsd redefinition for complex type personName
@XmlType(name="PersonName")
public class PersonName extends _PersonName {
    void setGeneration(Object value); Object getGeneration();
```

6.10.3 Bind Redefined Group Definition

The attribute or model group redefinition is used instead of the initial group definition to construct the property set of the content model(s) that reference the redefined attribute or model group definition. The construction of a property set is described in Section 6.3.2, “Java value class“.

Since there is no binding of an attribute or model group definition to a Java representation, no other special case handling is required for this binding.

6.11 Identity Constraint

An identity constraint does not represent any data, it represents a constraint that is enforced by validation. These constraints can be checked by optional validation that can be enabled at either unmarshal and/or marshal time.

6.12 Content Model - Particle, Model Group, Wildcard

This section describes the possible Java bindings for the content model of a complex type definition schema component with a *{content type}* property of `mixed` or `element-only`. The possible element content(s) and the valid ordering between those contents are constrained by the *{particles}* describing

the complex type definition's content model. The Java binding of a content model is realized by the derivation of one or more content-properties to represent the element content constrained by the model group. Section 6.12.1 through 6.12.7 describes the *element binding* of a content model.

6.12.1 Element binding style

The ideal Java binding would be to map each uniquely named element declaration occurring within a content model to a single JAXB property. The model group schema component constraint, element declarations consistent, specified in [XSD-Part 1] ensures that all element declarations/references having the same {target namespace} and {name} must have the same top-level type definition. This model allows the JAXB technology user to specify only the content and the JAXB implementation infers the valid ordering between the element content based on the *{particles}* constraints in the source schema. However, there do exist numerous scenarios that this ideal binding is not possible for parts of the content model or potentially the entire content model. For these cases, default binding has a fallback position of representing the element content and the ordering between the content using a *general content model*. The scenarios where one must fallback to the general content model will be identified later in this subsection.

6.12.2 Bind each element declaration name to a JAXB property

This approach relies on the fact that a model group merely provide constraints on the ordering between children elements and the user merely wishes to provide the content. It is easiest to introduce this concept without allowing for repeating occurrences of model groups within a content model. Conceptually, this approach presents all element declarations within a content model as a set of element declaration *{name}*'s. Each one of the *{name}*'s is mapped to a content-property. Based on the element content that is set by the JAXB application via setting content-properties, the JAXB implementation can compute the order between the element content using the following methods.

Computing the ordering between element content within **[children]** of an element information item

- Schema constrained fixed ordering or semantically insignificant ordering

The sequence in the schema represents an ordering between children elements that is completely fixed by the schema. Schema-constrained ordering is not exposed to the Java programmer when mapping each element in the sequence to a Java property. However, it is necessary for the marshal/unmarshal process to know the ordering. No new ordering constraints between children elements can be introduced by an XML document or Java application for this case. Additionally, the Java application does not need to know the ordering between children elements. When the compositor is `all`, the ordering between element content is not specified semantically and any ordering is okay. So this additional case can be handled the same way.

- Schema only constrains content and does not significantly constrain ordering

  If the ordering between the children elements is significant and must be accessible to the Java application, then the ordering is naturally preserved in Java representation via a collection. Below are examples where schema provides very little help in constraining order based on content.

```
<xs:choice maxOccurs="unbounded"> ... </xs:choice>
<xs:sequence maxOccurs="unbounded"> ... </xs:sequence>
```

6.12.3 General content property

A general content property is, as its name implies, the most general of all content properties. Such a property can be used with any content specification, no matter how complex. A general content property is represented as a List property as introduced in Section 5.5.2.2, “List Property,” on page 60. Unlike the prior approach where the JAXB implementation must infer ordering between the element content, this approach always requires the JAXB technology user to specify a valid ordering of element and text content. This approach has the benefit of providing the application with more control over setting and knowing the order between element content.

A general content property is capable of representing both element information items and character data items occurring within **[children]** of an element information item. Character data is inserted into the list as java.lang.String values. Element data is added to the list as instances of JAXB element. To support wildcard content occurring as part of a general content property, xml data content with no static Java binding is added and accessed from the list as instances of `org.w3c.org.dom.Node`.

The schema-derived Collection property getter method is annotated, either explicitly or by default mapping annotations, with the mapping annotations reflecting what content is within the Collection.

- If the content model is mixed, the property is annotated as `@XmlMixed`. See Section 6.12.4, “Bind mixed content“for details.
- Section 6.12.3.1, “Collection of Element types“describes an optimized binding of a collection of element values, instead of a collection of JAXB elements annotated with `@XmlElementRefs(@XmlElementRef, ...)`.
- If optimized binding can not be used, each element in the content model is represented by an `@XmlElementRef`, described in Section 6.7.5, “Bind to a Simple Element property“. If there is more than one element annotations needed, they must occur as elements in the map annotation `@XmlElementRefs` specified in Section 8.10.3, “@XmlElementRef”.

6.12.3.1 Collection of Element types

If the content model for a general content property meets all of the following constraints, the collection can be optimized to be a list of value classes instead of a list of JAXB elements.

- If the content model is not mixed and does not contain a wildcard.
- If none of the element declarations in the content model are abstract or the head of an element substitution group.
- If none of the element declarations in the content model have a xml datatype that is or derives from xs:list or xs:IDREF.
- For all element declarations in the content model, there does not exist two distinct element declarations whose types bind to the same Java datatype.
- If not more than one element declaration in the content model is nillable.

Such a collection is annotated with `@XmlElements` annotation, specified in Section 8, that contains a `@XmlElement` annotation for each unique Java datatype within the collection. The `@XmlElement` annotation associates an element name with each unique Java datatype in the collection

6.12.3.2 Examples

Example 1: Complex content model of Elements with primitive types

```
<xs:complexType name="Base">
    <xs:choice maxOccurs="unbounded">
        <xs:element name="A" type="xs:string"/>
        <xs:element name="B" type="xs:string"/>
        <xs:element name="C" type="xs:int"/>
    </xs:choice>
</xs:complexType>
public class ObjectFactory {
    // Element instance factories.
    JAXBElement<String> createBaseA(String value){...}
    JAXBElement<String> createBaseB(String value){...}
    JAXBElement<Integer> createBaseC(Integer value){...}
    // Type factories
    Base createBase(){...}
}
public class Base {

    /**
     * A general content list that can contain
     * element instances representing A, B and/or C.
     */
    @XmlElementRefs({@XmlElementRef(name="A", value=JAXBElement.class),
                     @XmlElementRef(name="B", value=JAXBElement.class),
                     @XmlElementRef(name="C", value=JAXBElement.class)})
    List<JAXBElement> getAOrBOrC(){...}
}
```

Example 2: Optimized Binding to a Collection of Element Types

XML Schema fragment:

```
<xs:complexType name="AType"/>
<xs:complexType name="BType"/>
<xs:complexType name="FooBar">
    <xs:choice maxOccurs="unbounded">
        <xs:element name="foo" type="AType"/>
        <xs:element name="bar" type="BType"/>
    </xs:choice>
</xs:complexType>
```

Default derived Java code:

```
public class AType { ... }
public class BType { ... }

class ObjectFactory {
    // element instance factories only
    JAXBElement<AType> createFooBarFoo(AType value);
    JAXBElement<BType> createFooBarBar(BType value);
}

public class FooBar {
    /**
    '* Collection of element types: AType and BType. */
    @XmlElements({@XmlElement(value=AType.class, name="Foo"),
                  @XmlElement(value=BType.class, name="Bar")})
    List<Object> getFooOrBar() {...}
};
```

6.12.4 Bind mixed content

When a complex type definition's *{content type}* is "mixed," its character and element information content is bound to general content list as described in Section 6.12.3, "General content property." Character information data is inserted as instances of `java.lang.String` into a JAXB collection property.

The schema-derived Collection property getter method is annotated, either explicitly or by default mapping annotations, with the mapping annotation `@XmlMixed`, specified in Section 8.

Example:

Schema fragment loosely derived from mixed content example from [XSD Part 0].

```
<xs:element name="letterBody">
    <xs:complexType mixed="true">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="quantity" type="xs:positiveInteger"/>
            <xs:element name="productName" type="xs:string"/>
            <!-- etc. -->
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Derived Java code:

```
import java.math.BigInteger;
class ObjectFactory {
    // element instance factories only
    JAXBElement<LetterBody> createLetterBody(LetterBody value);
    JAXBElement<String> createLetterBodyName(String value);
    JAXBElement<BigInteger>
                  createLetterBodyQuantity(BigInteger value);
    JAXBElement<String>
                  createLetterBodyProductName(String value);
}
public class LetterBody implements JAXBElement<LetterBody> {

    /** Mixed content can contain instances of Element classes
        Name, Quantity and ProductName. Text data is represented as
        java.util.String for text.
    */
    @XmlMixed
    @XmlElementRefs({
        @XmlElementRef(name="productName", type=JAXBElement.class),
        @XmlElementRef(name="quantity", type=JAXBElement.class),
        @XmlElementRef(name="name", type=JAXBElement.class)})
    List getContent(){...}
}
```

The following instance document

```
<letterBody>
Dear Mr.<name>Robert Smith</name>
Your order of <quantity>1</quantity> <productName>Baby
Monitor</productName> shipped from our warehouse. ....
</letterBody>
```

could be constructed using JAXB API.

```
LetterBody lb = ObjectFactory.createLetterBody(null);
List gcl = lb.getContent();
gcl.add("Dear Mr.");
gcl.add(ObjectFactory.createLetterBodyName("Robert Smith"));
gcl.add("Your order of ");
gcl.add(ObjectFactory.
          createLetterBodyQuantity(new BigInteger("1")));
gcl.add(ObjectFactory.createLetterBodyProductName("Baby Monitor"));
gcl.add("shipped from our warehouse");
```

Note that if any element instance is placed into the general content list, *gcl*, that is not an instance of LetterBody.Name, LetterBody.Quantity or LetterBody.ProductName, validation would detect the invalid content model. With the fail fast customization enabled, element instances of the wrong type are detected when being added to the general content list, *gcl*.

6.12.5 Bind wildcard schema component

A wildcard is mapped to a simple content-property with:

- Content-property *name* set to the constant "`any`". A binding schema customization could provide a more semantically meaningful content-property name.
- Content-property *base type* set to `java.lang.Object` by default.

  Wildcard content is represented as one of the following:

  a. JAXB element
  Either an instance of `javax.xml.bind.JAXBElement<T>` or a JAXB class annotated with `@XmlRootElement`.
  Corresponds to a recognized global element tag name registered with the instance `javax.xml.bind.JAXBContext`, meaning the schema(s) describing the element content is registered with the `JAXBContext` instance, see Section 4.2, "JAXBContext," on page 32 on how bindings are registered with a `JAXBContext` instance.,

  b. instance of `javax.xml.bind.JAXBElement`.
  Corresponds to an unknown element name but a recognized type definition specified by ***@xsi:type*** on the element. JAXBElement *declaredType* is set to `java.lang.Object` since the unknown element declaration's default type is `xs:anyType`.

c. element node instance of a supported xml infoset API
Necessary to represent Xml data content that does not have a schema defined element or type definition. Such content is allowed by element **xs:any** with attribute **@processContents="lax"** or **`"skip"`**.

- See content-property predicate for a wildcard.
- If the `maxOccurs` is greater than one, the content property is mapped to a collection property. The default collection property is a List property of base type java.lang.Object.
- These is no *default value*.

Since the schema does not contain any information about the element content of a wildcard content, even the content-property, by default, can not infer an XML element tag for wildcard element content.

The schema-derived property getter method for representing wildcard content is annotated, either explicitly or by default mapping annotations, with the mapping annotation `@XmlAnyElement`, specified in Section 8. The @XmlAnyElement annotation element values are derived in terms of the abstract model properties for wildcard summarized in Section F.1.10, "Wildcard Schema Component," on page 351 as follows:

Table 6-1 Annotate JAXB property with @XmlAnyElement element-value pairs

@XmlAnyElement element	@XmlAnyElement element value
lax	If wildcard schema component's *{process contents}* is `lax` or `strict`, set `@XmlAnyElement.lax()` to `true`. else if *{process contents}* is `skip`, set `@XmlAnyElement.lax()` to `false`.
value	`javax.xml.bind.annotation.W3CDomHandler.class`

6.12.6 Bind a repeating occurrence model group

A choice or sequence model group, containing more than one member, with a repeating occurrence, `maxOccurs` attribute greater than one, is bound to a general content property in the following manner:

- Content-property *name* is derived in following ways:
  - If a named model group definition is being referenced, the value of its *{name}* property is mapped to a Java identifier for a method using the algorithm specified in Section D.2, "The Name to Identifier Mapping Algorithm," on page 333.
  - To derive a content property *name* for unnamed model group, see Section D.4, "Deriving an identifier for a model group," on page 339.
- Content-property *base type* set to `java.lang.Object`. A binding schema customization could provide a more specialized java class.

- Content-property *predicate* validates the order between element instances in the list and whether the occurrence constraints for each element instance type is valid according to the schema.
- Since the `maxOccurs` is always greater than one, the content property is mapped to a collection property. The default collection property is a List property.
- These is no *default value*.

The schema-derived collection property is annotated as specified in Section 6.12.3, "General content property"and Section 6.12.3.1, "Collection of Element types".

Local structural Constraints

The list content property's value must satisfy the content specification of the model group. The ordering and element contents must satisfy the constraints specified by the model group.

6.12.7 Content Model Default Binding

The following rules define *element* binding style for a complex type definition's content model.

1. If *{content type}* is mixed, bind the entire content model to a general content property with the content-property name "`content`". See Section 6.12.4, "Bind mixed content" for more details.
2. **If (1) a particle has *{max occurs}* >1 and (2) its *{term}* is a model group and (3) all the particles in the model group have {terms} that bind to different Java datatypes, bind to a collection of element types. See complete list of constraints required to perform this optimized binding in Section 6.12.3.1, "Collection of Element types".**
3. **If (1) a particle has *{max occurs}* >1 and (2) its *{term}* is a model group, then that particle and its descendants are mapped to one general content property that represents them. See Section 6.12.6, "Bind a repeating occurrence model group" for details.**
4. **Process all the remaining particles (1) whose *{term}* are wildcard particles and (2) that did not belong to a repeating occurrence model group bound in step. 2. If there is only one wildcard, bind it as specified in Section 6.12.5, "Bind wildcard schema component." If**

there is more than one, then fallback to representing the entire content model as a single general content property. See Section 6.12.3, "General content property).

5. **Process all particles (1) whose *{term}* are element declarations and (2) that do not belong to a repeating occurrence model group bound in step.2.**

   First, we say a particle has a label *L* if it refers to an element declaration whose *{name}* is *L*. Then, for all the possible pair of particles *P* and *P'* in this set, if the following constraints are not met:

   a. If *P* and *P'* have the same label, then they must refer to the same element declaration.

   b. If *P* and *P'* refer to the same element reference, then its closest common ancestor particle may not have sequence as its *{term}*.

   If either of the above constraints are violated,-it is not possible to map each element declaration to a unique content property. Fallback to representing the entire content model as a single general content property.

   Otherwise, create a content property for each label *L* as follows:

   - The content property *name* is derived from label name *L*.
   - The *base type* will be the Java type to which the referenced element declaration maps.
   - The content property *predicate* reflects the occurrence constraint.
   - The content property *collection type* defaults to '`list`' if there exist a particle with label *L* that has *{maxOccurs}* > 1.
   - For the default value, if all particles with label *L* has a *{term}* with the same *{value constraint}* default or fixed value, then this value. Otherwise none.

Below is an example demonstrating of not meeting the uniqueness constraints of 5(a) and 5(b) specified above.

```
<xs:sequence>
    <xs:choice>
        <xs:element ref="ns1:bar"/> (A)
        <xs:element ref="ns2:bar"/> (B)
    </xs:choice>
    <xs:element ref="ns1:bar"/> (C)
</xs:sequence>
```

The pair *(A,B)* violates the first clause because they both have the label “bar” but they refer to different element declarations. The pair *(A,C)* violates the second clause because their nearest common ancestor particle is the outermost `<sequence>`. This model group fragment is bound to a general content property.

6.12.7.1 Default binding of content model “derived by extension”

If a content-property naming collision occurs between a content-property that exists in an base complex type definition and a content-property introduced by a “derive by extension” derived complex type definition, the content-properties from the colliding property on are represented by a general content property with the default property name `rest`.

Example:
derivation by extension content model with a content-property collision.

Given XML Schema fragment:

```
<xs:complexType name="Base">
    <xs:sequence>
        <xs:element name="A" type="xs:int"/>
        <xs:element name="B" type="xs:int"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="Derived">
    <xs:complexContent>
        <xs:extension base="Base">
            <xs:sequence>
                <xs:element name="A" type="xs:int"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Default binding derived Java code:

```
public class Base {
    int getA(){...} void setA(int){...}
    int getB(){...} void setB(int){...}
}

public class Derived extends Base {
    /**
     * Instances of Derived.A must be placed in this general
     * content propert that represents the rest of the content
     * model. [7] */
    List getRest(){...}
}
class ObjectFactory {
    // element instance factories only
    JAXBElement<Integer> createDerivedA(Integer value){...}
}
```

6.12.7.2 Bind single occurrence choice group to a choice content property

Setting the `choiceContentProperty` attribute of `<jaxb:globalBindings>` as specified in Section 7.5.1, “Usage,” on page 166 enables this customized binding option.

A non-repeating choice model group is bound to a simple property. The simple choice content property is derived from a choice model group as follows:

- The choice content property name is either the referenced model group definition *{name}* or obtained using the algorithm specified in Section D.4, “Deriving an identifier for a model group,” on page 339.
- The choice content property `base type` is the first common supertype of all items within the choice model group, with `java.lang.Object` always being a common root for all Java objects.[8]

---

7. Specifying a customization of the local element declaration A within Derived complex type to a different property name than A would avoid the fallback position for this case.

8. Note that primitive Java types must be represented by their Java wrapper classes when *base type* is used in the choice content property method signatures. Also, all sequence descendants of the choice are treated as either a general content property or are mapped to their own value class.

- The predicate
- The collection type defaults to List if one or more items in the choice model group bind to List.
- No default value.

A choice property consists of the following methods:

- The `getChoiceID` method returns the set value. If the property has no set value then the value `null` is returned. Note that a set value of a primitive Java type is returned as an instance of the corresponding Java wrapper class.
- The `setChoiceID` method has a single parameter that is the type of the choice content property `base type`.

The `globalBindings` and property customization attribute, `choiceContentProperty`, enables this customized binding. The customization is specified in Section 7.5, "`<globalBindings>` Declaration."

Example:

XML Schema representation of a choice model group.

```
<xs:choice>
    <xs:element name="foo" type="xs:int"/>
    <xs:element name="bar" type="xs:string"/>
</xs:choice>
```

Derived choice content property method signatures:

```
void setFooOrBar(Object){...}
Object getFooOrBar(){...}
```

6.13 Modifying Schema-Derived Code

There exist a number of use cases on why a developer would find it beneficial to modify schema-derived classes. Here are some of those use cases.

- Add functionality to schema-derived classes.
  Since schema-derived classes are derived from a data description

language, the derived classes only represent data and have no object-level functionality.

- Add polymorphic methods to Java class hierarchy generated from XML Schema type definition derivation hierarchy.
- Initialize a JAXB property or field representing an XML element with a default value. Regretfully, XML Schema element defaulting is insufficient to accomplish this. Note that XML Schema attribute defaulting is sufficient and does not require this approach.

The JAXB 2.0 schema-derived class was designed to be easily understandable and modifiable by a developer. For many development environments, it is not sufficient to only run the schema compiler once due to modification of the schema-derived classes. Since schemas evolve over time, it is desirable to have the ability to regenerate schema-derived classes from an updated schema while preserving modification made by a developer. Given the complexities of supporting this capability, a JAXB implementation is not required to support regeneration from a schema into previously modified schema-derived classes. External tools, such as an IDE, could assist in supporting the sophisticated task of regeneration of a modified schema-derived class in the future. To enable tools to support regeneration, a JAXB implementation is required to have an option for generating an annotation that enables a portable means for distinguishing between developer code and generated code in a schema-derived class.The next section describes the portable format for distinguishing between generated and developer added/modified methods and /or fields in a schema-derived class.

6.13.1 Distinguish between generated and user added code

A schema compiler must have an option to generate the Common Annotation, `@javax.annotation.Generated` annotation, specified in [CA], on every generated class, method and field. If a developer does modify an `@Generated` annotated method or field, they must denote this modification by deleting the `@Generated` annotation. If a developer adds a new method or field, it will not have an `@Generated` annotation on it. Based on these conventions, a JAXB implementation in conjunction with an IDE or other external tool, would be able to support regeneration of schema-derived code while preserving developer additions/modifications to methods and fields in a schema-derived class.

When schema compiler option to generate `@Generated` annotation is selected, the table describes the annotation to be generated.

Table 6-1 Annotate generated class, field and property with @Generated element-value pairs

@Generated element	@Generated element value
value	fully qualified class name of schema compiler
date	date of generation of schema-derived class. Value must follow the ISO 8601 standard.
comment	`optional. Is implementation specific.`

6.14 Default Binding Rule Summary

Note that this summary is non-normative and all default binding rules specified previously in the chapter take precedence over this summary.

- Bind the following to Java package:
  - XML Namespace URI
- Bind the following XML Schema components to Java value class:
  - Named complex type
- Bind to typesafe enum class:
  - A named simple type definition with a basetype that derives from "`xs:NCName`" and has enumeration facets.
- Bind the following XML Schema components to an element instance factory that returns `javax.xml.bind.JAXBElement<T>`
  - A global element declaration with a named type definition.
  - Local element declaration with a named type definition that can be inserted into a general content list.
- Bind the following XML Schema components to a Java Element class
  - A global element declaration with anonymous type definition to a Java value class.

    - Local element declaration with anonymous type definition that can be inserted into a general content list.
- Bind to Java property
    - Attribute use
    - Particle with a term that is an element reference or local element declaration.

      Additionally, generate an element property for an element reference to the head element of a substitution group.
- Bind to JAXB property:
  `getOtherAttributes(): java.util.Map<QName, String>`
    - Attribute Wildcard occurring directly or indirectly via an attribute group reference in a complex type definition.
- Bind model group and wildcard content with a repeating occurrence and complex type definitions with `mixed` *{content type}* to:
    - A general content property - a List content-property that holds Java instances representing element information items and character data items. To support dynamic Xml content that validates against xs:any processContents=”lax” or “skip”, allow instances of org.w3c.dom.Node into the list.

Table 6-2 Summarize default XSD to Java binding for Figure 5.4 and Figure 5.5

XML Schema	**Java Representation**
Schema targetNamespace	Package
Global Element Declaration with named type definition	ObjectFactory.elementInstanceFactory method returning JAXBElement<T>
Global Complex Type Definition (Named)	value class/class + ObjectFactory.typeInstanceFactory method
Global Simple Type Definition -derive base of string - has @enum facet(s)	enum type
SimpleType facets	ConstraintPredicate

Table 6-2 Summarize default XSD to Java binding for Figure 5.4 and Figure 5.5

XML Schema	Java Representation
Attribute Uses Local Element Declaration	Property
facet **@maxOccurs > 1** xsd:list	PropertyStyle List
@fixed PropertyStyle	Constant
Global Element Declaration with anonymous type definition	value class for anonymous type+ ObjectFactory.typeInstanceFactory+ ObjectFactory.elementInstanceFactory method
Element reference to SubstitutionGroup Head maxOccurs = “1”	Simple + Element property
Element reference to SubstitutionGroup Head maxOccurs > “1”	List<JAXBElement<T>>

CHAPTER 7

CUSTOMIZING XML SCHEMA TO JAVA REPRESENTATION BINDING

The default binding of source schema components to derived Java representation by a binding compiler sometimes may not meet the requirements of a JAXB application. In such cases, the default binding can be customized using a *binding declaration*. Binding declarations are specified by a *binding language*, the syntax and semantics of which are defined in this chapter.

All JAXB implementations are required to provide customization support specified here unless explicitly stated as optional.

7.1 Binding Language

The binding language is an XML based language which defines constructs referred to as *binding declarations*. A binding declaration can be used to customize the default binding between an XML schema component and its Java representation.

The schema for binding declarations is defined in the namespace `http://java.sun.com/xml/ns/jaxb`. This specification uses the namespace prefix "`jaxb`" to refer to the namespace of binding declarations. For example,

```
<jaxb: binding declaration>
```

A binding compiler interprets the binding declaration relative to the source schema and a set of default bindings for that schema. Therefore a source schema

need not contain a binding declarations for every schema component. This makes the job of a JAXB application developer easier.

There are two ways to associate a binding declaration with a schema element:

- as part of the source schema (*inline annotated schema*)
- external to the source schema in an *external binding declaration.*

The syntax and semantics of the binding declaration is the same regardless of which of the above two methods is used for customization.

A binding declaration itself does not identify the schema component to which it applies. A schema component can be identified in several ways:

- explicitly - e.g. QName, XPath expressions etc.
- implicitly - based on the context in which the declaration occurs.

It is this separation which allows the binding declaration syntax to be shared between inline annotated schema and the external binding.

7.1.1 Extending the Binding Language

In recognition that there will exist a need for additional binding declarations than those currently specified in this specification, a formal mechanism is introduced so all JAXB processors are able to identify *extension binding declarations*. An extension binding declaration is not specified in the *jaxb:* namespace, is implementation specific and its use will impact portability. Therefore, binding customization that must be portable between JAXB implementations should not rely on particular customization extensions being available.

The namespaces containing extension binding declarations are specified to a JAXB processor by the occurrence of the global attribute `<jaxb:extensionBindingPrefixes>` within an instance of `<xs:schema>` element. The value of this attribute is a whitespace-separated list of namespace prefixes. The namespace bound to each of the prefixes is designated as a customization declaration namespace. Prefixes are resolved on the `<xs:schema>` element that carries this attribute. It is an error if the prefix fails to resolve. This feature is quite similar to the extension-element-prefixes attribute in [XSLT 1.0] `http://www.w3.org/TR/xslt10/#extension`, introduces extension namespaces for extension instructions and functions for XSLT 1.0.

This specification does not define any mechanism for creating or processing extension binding declarations and does not require that implementations support any such mechanism. Such mechanisms, if they exist, are implementation-defined.

7.1.2 Inline Annotated Schema

This method of customization utilizes on the <appinfo> element specified by the XML Schema [XSD PART 1]. A binding declaration is embedded within the <appinfo> element as illustrated below.

```
<xs:annotation>
    <xs:appinfo>
        <binding declaration>
    </xs:appinfo>
</xs:annotation>
```

The inline annotation where the binding declaration is used identifies the schema component.

7.1.3 External Binding Declaration

The external binding declaration format enables customized binding without requiring modification of the source schema. Unlike inline annotation, the remote schema component to which the binding declaration applies must be identified explicitly. The <jaxb:bindings> element enables the specification of a remote schema context to associate its binding declaration(s) with. Minimally, an external binding declaration follows the following format.

```
<jaxb:bindings [schemaLocation = "xs:anyURI"]>
    <jaxb:bindings [node = "xs:string"]>*
        <binding declaration>
    <jaxb:bindings>
</jaxb:bindings>
```

The *schemaLocation* attribute is optional for specifying <jaxb:globalBindings>, and The *node* attribute is optional for specifying <jaxb:schemaBindings>. The attributes *schemaLocation* and *node* are used to construct a reference to a node in a remote schema. The binding declaration is applied to this node by the binding compiler as if the

binding declaration was embedded in the node's <xs:appinfo> element. The attribute values are interpreted as follows:

- *schemaLocation* - It is a URI reference to a remote schema.
- *node* - It is an XPath 1.0 expression that identifies the schema node within schemaLocation to associate binding declarations with.

An example external binding declaration can be found in Section E.1, "Example."

7.1.3.1 Restrictions

- The external binding element <jaxb:bindings> is only recognized for processing by a JAXB processor when its parent is an <xs:appinfo> element, it is an ancestor of another <jaxb:bindings> element, or when it is root element of a document. An XML document that has a <jaxb:bindings> element as its root is referred to as an external binding declaration file.
- The top-most <jaxb:binding> element within an <xs:appinfo> element or the root element of an external binding file must have its schemaLocation attribute set.

7.1.4 Version Attribute

The normative binding schema specifies a global version attribute. It is used to identify the version of the binding declarations. For example, a future version of this specification may use the version attribute to specify backward compatibility. To indicate this version of the specification, the version should be "2.0". It is also valid for @version to be "1.0". If any other version is specified, it must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations."

The version attribute must be specified in one of the following ways:

- If customizations are specified in inline annotations, the version attribute must be specified in <xs:schema> element of the source schema. For example,

```
<xs:schema jaxb:version="2.0">
```

- If customizations are specified in an external binding file, then the jaxb:version attribute must be specified in the root element

`<jaxb:bindings>` in the external binding file. Alternately, a local `version` attribute may be used. Thus the version can be specified either as

```
<jaxb:bindings version="2.0" ... />
```

or

```
<jaxb:bindings jaxb:version="2.0" ... />
```

Specification of both `version` and `<jaxb:version>` must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations."

7.1.5 Invalid Customizations

A *non conforming* binding declaration is a binding declaration in the `jaxb` namespace but does not conform to this specification. A non conforming binding declaration results in a *customization error.* The binding compiler must report the customization error. The exact error is not specified here. For additional requirements see Chapter 9, "Compatibility."

The rest of this chapter assumes that non conforming binding declarations are processed as indicated above and their semantics are not explicitly specified in the descriptions of individual binding declarations.

7.2 Notation

The source and binding-schema fragments shown in this chapter are meant to be illustrative rather than normative. The normative syntax for the binding language is specified in Appendix , "Normative Binding Schema Syntax." in addition to the other normative text within this chapter. All examples are non-normative.

- Metavariables are in *italics*.
- Optional attributes are enclosed in `[ square="bracket" ]`.
- Optional elements are enclosed in `[ <elementA> ... </ elementA> ]`.

- Other symbols: ‘`,`” denotes a sequence, ‘`|`’ denotes a choice, ‘+’ denotes one or more, ‘`*`’ denotes zero or more.
- The prefix `xs:` is used to refer to schema components in W3C XML Schema namespace.
- In examples, the binding declarations as well as the customized code are shown in bold like this: **<appinfo> <annotation>** or **getAddress()**.

7.3 Naming Conventions

The naming convention for XML names in the binding language schema are:

- The first letter of the first word in a multi word name is in lower case.
- The first letter of every word except the first one is in upper case.

For example, the XML name for the Java property basetype is baseType.

7.4 Customization Overview

A binding declaration customizes the default binding of a schema element to a Java representation. The binding declaration defines one or more *customization values* each of which customizes a part of Java representation.

7.4.1 Scope

When a customization value is defined in a binding declaration, it is associated with a *scope*. A scope of a customization value is the set of schema elements to which it applies. If a customization value applies to a schema element, then the schema element is said to be *covered* by the scope of the customization value. The scopes are:

- **global scope:** A customization value defined in `<globalBindings>` has *global scope*. A global scope covers all the schema elements in the source schema and (recursively) any schemas that are included or imported by the source schema.

- **schema scope:** A customization value defined in `<schemaBindings>` has *schema scope*. A schema scope covers all the schema elements in the target namespace of a schema.
- **definition scope:** A customization value in binding declarations of a type definition or global declaration has *definition scope.* A definition scope covers all schema elements that reference the type definition or the global declaration. This is more precisely specified in the context of binding declarations later on in this chapter.
- **component scope:** A customization value in a binding declaration has *component scope* if the customization value applies only to the schema element that was annotated with the binding declaration.

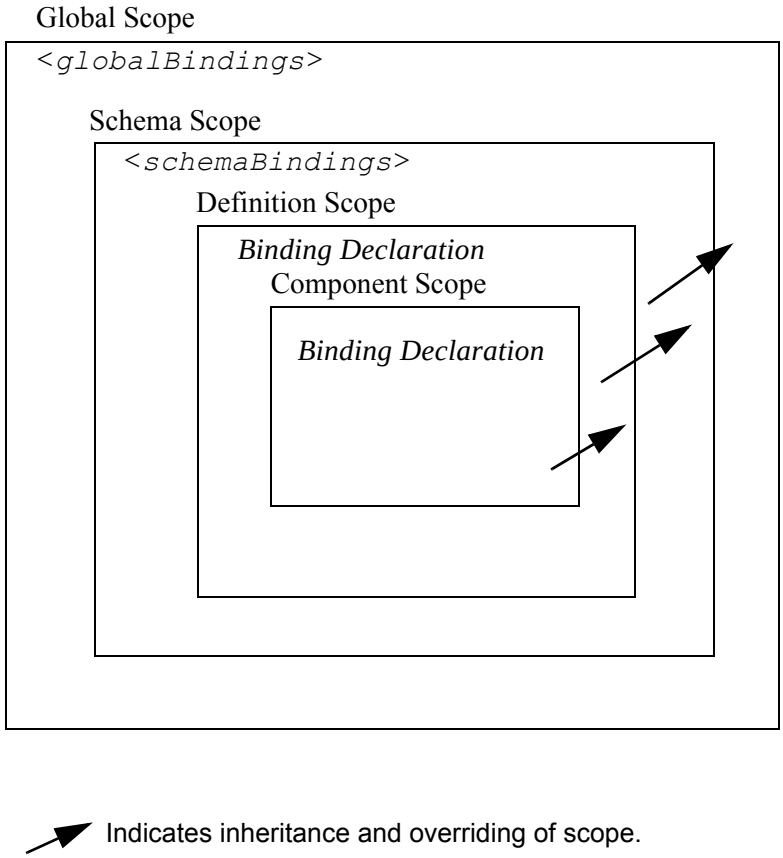


Figure 7.1 Scoping Inheritance and Overriding For Binding Declarations

The different scopes form a taxonomy. The taxonomy defines both the inheritance and overriding semantics of customization values. A customization value defined in one scope is inherited for use in a binding declaration covered by another scope as shown by the following inheritance hierarchy:

- a schema element in schema scope inherits a customization value defined in global scope.
- a schema element in definition scope inherits a customization value defined in schema or global scope.
- a schema element in component scope inherits a customization value defined in definition, schema or global scope.

Likewise, a customization value defined in one scope can override a customization value inherited from another scope as shown below:

- value in schema scope overrides a value inherited from global scope.
- value in definition scope overrides a value inherited from schema scope or global scope.
- value in component scope overrides a value inherited from definition, schema or global scope.

7.4.2 XML Schema Parsing

Chapter 5 specified the bindings using the abstract schema model. Customization, on the other hand, is specified in terms of XML syntax not abstract schema model. The XML Schema specification [XSD PART 1] specifies the parsing of schema elements into abstract schema components. This parsing is assumed for parsing of annotation elements specified here. In some cases, [XSD PART 1] is ambiguous with respect to the specification of annotation elements. Section 7.15, “Annotation Restrictions” outlines how these are addressed.

Design Note – The reason for specifying using the XML syntax instead of abstract schema model is as follows. For most part, there is a one-to-one mapping between schema elements and the abstract schema components to which they are bound. However, there are certain exceptions: local attributes and particles. A local attribute is mapped to two schema components: {attribute declaration} and {attribute use}. But the XML parsing process associates the annotation with the {attribute declaration} not the {attribute use}. This is tricky and not obvious. Hence for ease of understanding, a choice was made to specify customization at the surface syntax level instead.

7.5 <globalBindings> Declaration

The customization values in "<globalBindings>" binding declaration have global scope. This binding declaration is therefore useful for customizing at a global level.

7.5.1 Usage

```
<globalBindings
    [ collectionType = "collectionType" ]
    [ fixedAttributeAsConstantProperty= "true" | "false" | "1" | "0"
]
    [ generateIsSetMethod= "true" | "false" | "1" | "0" ]
    [ enableFailFastCheck = "true" | "false" | "1" | "0" ]
    [ choiceContentProperty = "true" | "false" | "1" | "0" ]
    [ underscoreBinding  = "asWordSeparator" | "asCharInWord" ]
    [ typesafeEnumBase = "typesafeEnumBase" ]
    [ typesafeEnumMemberName = "skipGeneration" |
                              "generateName" | "generateError" ]
    [ typesafeEnumMaxMembers = "xxxx"]
    [ enableJavaNamingConventions = "true" | "false" | "1" | "0" ]
    [ generateElementClass = "false" | "true" | "0" | "1" ]
    [ generateElementProperty = "false" | "true" | "0" | "1" ]
    [ generateValueClass = "true" | "true" | "0" | "1" ]
    [ optionalProperty = "wrapper" | "primitive" | "isSet" ]
    [ mapSimpleTypeDef = "true" | "false" | "1" | "0" ]
    [ localScoping = "nested" | "toplevel" ] >
    [ <javaType> ... </javaType> ]*
    [ <serializable uid="xxxx"/> ]*
</globalBindings>
```

The following customization values are defined in global scope:

- *collectionType* if specified, must be either "indexed" or any fully qualified class name that implements *java.util.List.* The default value is to any fully qualified class name that implements *java.util.List*.
- *fixedAttributeAsConstantProperty* if specified , defines the customization value *fixedAttributeAsConstantProperty*. The value must be one of "true", false", "1" or"0". The default value is "false".
- *generateIsSetMethod* if specified, defines the customization value of *generateIsSetMethod.* The value must be one of "true", false", "1" or"0". The default value is "false". Consider customizing using the newly introduced *optionalProperty* before using this JAXB 1.0 customization.
- *enableFailFastCheck* if specified, defines the customization value *enableFailFastCheck.* The value must be one of "true",

"false", "1" or"0". If *enableFailFastCheck* is "true" or "1" and the JAXB implementation supports this optional checking, type constraint checking when setting a property is performed as described in Section 5.5, "Properties". The default value is "false".

- *choiceContentProperty* if specified,defines the customization value *choiceContentProperty* . The value must be one of "true", false", "1" or"0". The default value is "false".
- *underscoreBinding* if specified, defines the customization value *underscoreBinding*. The value must be one of "asWordSeparator" or "asCharInWord". The default value is "asWordSeparator".
- *enableJavaNamingConventions* if specified, defines the customization value *enableJavaNamingConventions*. The value must be one of "true", false", "1" or"0". The default value is "true".
- *typesafeEnumBase* if specified, defines the customization value *typesafeEnumBase.* The value must be a list of QNames, each of which must resolve to a simple type definition. Only simple type definitions with an enumeration facet and a restriction base type listed in *typesafeEnumBase* or derived from a type listed in *typesafeEnumBase* is bound to a *typesafeEnumClass* by default as specified in Section 6.2.3, "Enum Type". The default value of *typesafeEnumBase* is "xs:string".

  The *typesafeEnumBase* cannot contain the following simple types and therefore a JAXB implementation is not required to support the binding of the these types to typesafe enumeration class: *"xs:QName", "xs:NOTATIION", xs:base64Binary", "xs:hexBinary", "xs:date", "xs:time", "xs:dateTime", "xs:duration", "xs:gDay", "xs:gMonth", "xs:gYear", "xs:gMonthDay", "xs:gYearMonth", "xs:IDREF", "xs:ID".* If any of them are specified, it must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations." JAXB implementation must be capable of binding any other simple type listed in *typesafeEnumBase* to a typesafe enumeration class.
- *typesafeEnumMemberName* if specified, defines the customization value *typesafeEnumMemberName.* The value must be one of skipGeneration, generateError or generateName. The

default value is `skipGeneration`. See Section 7.5.5, "@typesafeEnumMemberName" for details.

- *typesafeEnumMaxMembers* if specified, defines the maximum number of enum facets that a simple type definition can have and be consider to binding to an enum type by default. The attributes type is `xs:int` and its default value is `256`.
- *`generateElementClass`* if specified as true, a schema-derived Element class, as specified in Section 5.6.2, "Java Element Class", is generated for each element declaration that has an element factory method generated for it. Its default value is false.
- *`generateElementProperty`* if specified as true, controls the generation of JAXBElement property. The value must be one of "true", "false", "1", or "0". The default is absence of the value.
- *`generateValueClass`* if specified as true, a schema-derived Java value class is generated for each complex type definiton.Value class is specified in Section 5.4.1, "Value Class," on page 53. If generateValueClass is specified as false, a schema-derived interface and implementation class is generated for each complex type definition as specified in Section 5.4.2, "Java Content Interface". The attribute's default value is true. See examples of this binding in Section 7.5.4, "generateElementClass and generateValueClass".
- zero or more *`javaType`* binding declarations. Each binding declaration must be specified as described in Section 7.9, "`<javaType>` Declaration," on page 197."
- zero or one serializable binding declaration.
- *`optionalProperty`* controls how a JAXB property with a primitive base type that represents an optional non-nillable element/ attribute is bound. If the attribute has the value "wrapper", then the base type of the JAXB property is the wrapper class for the primitive type. A user can indicate that this optional property is not set by calling the setter with "null" value. If the attribute's value is "primitive", it binds as it did in JAXB 1.0. If the attribute's value is "isSet", it binds the optional property using the primitive base type and also the isSet/unset methods are generated for the optional property. The attribute's default value is "wrapper".
- *mapSimpleTypeDef* controls whether a JAXB mapped class should be generated for each simple type definition as specified in Section 6.2.2.2, "Bind to a JAXB mapped class". This attribute's default value is `false`. This customization eases preserving simple type substituting precisely as

described in Section 6.7.4.2, “Type Substitution of a Simple Type Definition”.

- *localScoping* attribute can have the value of either *nested* or *toplevel*. This attribute describes the JAXB binding of nested XML schema component to either a *nested* schema-derived JAXB class or a *toplevel* schema-derived JAXB class. To avoid naming collisions between nested components, the default value for this attribute is *nested*. A developer can customize *localScoping* to *toplevel* w hen schema components nest too deeply or an application would prefer to not work with nested classes.

The semantics of the above customization values, if not specified above, are specified when they are actually used in the binding declarations.

For inline annotation, a `<globalBindings>` is valid only in the annotation element of the `<schema>` element. There must only be a single instance of a `<globalBindings>` declaration in the annotation element of the `<schema>` element.

7.5.2 Customized Name Mapping

A customization value can be used to specify a name for a Java object (e.g. class name, package name etc.). In this case, a customization value is referred to as a *customization name.*

A customization name is always a legal Java identifier (this is formally specified in each binding declaration where the name is specified). Since customization deals with customization of a Java representation to which an XML schema element is bound, requiring a customization name to be a legal Java identifier rather than an XML name is considered more meaningful.

A customization name may or may not conform to the recommended Java language naming conventions. [JLS - Java Language Specification, Second Edition, Section 6.8, “Naming Conventions”]. The customization value *enableJavaNamingConventions* determines if a customization name is mapped to a Java identifier that follows Java language naming conventions or not.

If *enableJavaNamingConventions* is defined and the value is `"true"` or `"1"`, then the customization name (except for constant name) specified in the section from where this section is referenced must be mapped to Java identifier which follows the Java language naming conventions as specified in Section D.6, “Conforming Java Identifier Algorithm”; otherwise the customized name must be used as is.

7.5.3 Underscore Handling

The **[jaxb:globalBindings]** attribute customization *underscoreBinding* allows for the preservation of underscore(s) occurring in an XML name when deriving a a Java identifier from it.

The default value for *@underscoreBinding* is `"asWordSeparator"` and categorizes underscore, '_', as a punctuation mark in the XML name to Java identifier algorithm specified in Appendix D.2. The resulting algorithm transforms one or more consecutive underscores in an XML name to camel case separated words in the derived Java class and method names. Examples of this mapping are in Table D-2.

When *@underscoreBinding* is `"asCharInWord",` underscore ('_') is considered a special letter within a word. The result is that all underscore characters from the original XML name are preserved in the derived Java identifier. Example of this mapping are in Table D-3.

7.5.4 generateElementClass and generateValueClass

The following code examples illustrate default binding to value class and customization to bind to interface/implementation classes.

CODE EXAMPLE 7-1 Default Binding to a value class.

Schema fragment:

```
<xs:complexType name="USAddress">
    <xs:attribute name="City" type="xs:string"/>
</xs:complexType>
```

Default Value Class:

```
public class USAddress {
    public USAddress() {...}
    public String getCity() {....}
    public void setCity(String value) {...}
    ...
}
```

Customization `<jaxb:globalBinding generateValueClass="false">` generates following interface instead of default value class:

CODE EXAMPLE 7-2 Customized binding to an interface.

```
public interface USAddress {
    String getCity();
    void setCity(String value);
}
```

CODE EXAMPLE 7-3 Generation of an Element Class

Schema fragment:

```
<xs:element name=”Address” type=”USAddress”/>

// Default Java binding of global element to element instance factory
public ObjectFactory {
        JAXBElement<USAddress> createAddress(USAddress value);
}
```

`<jaxb:globalBinding generateElementClass=”true”/>` results in generation of following Element class:

```
public class Address extends JAXBElement<USAddress> {
}
```

7.5.5 @typesafeEnumMemberName

.If there is a collision among the generated constant fields **name** or if it is not possible to generate a legal Java identifier for one or more of the generated constant field names, then the binding is determined based on the value of *@typesafeEnumMemberName* of element **[jaxb:globalBindings]**.

- *skipGeneration*
  An enum type is not generated. This is the default behavior if *typesafeEnumMemberName* has not been specified. A binding compiler may report a warning on why the simple type definition was not bound to an enum type.
- *generateName*
  The constant fields **name** is “*VALUE_<N>*" where *namet360492f eame*

7.5.6 <serializable> Declaration

When the serializable customization is specified, all schema-derived classes implement `java.io.Serializable`. Each class is generated with a `serialVersionUID` field set to the value specified by *@uid*.

```
private static final long serialVersionUID = <value of @uid>;
```

The JAXB user is required to identify when schema-derived classes do not follow Java serialization class evolution rules and change the generated `serialVersionUID` field by changing the **[serializable]** element's attribute *@uid* value.

7.5.7 @generateElementProperty

Some schemas use both minOccurs="0" on element as well as nillable="true", causing the generation of JAXBElement. This customization lets you control this behavior. This attribute may take two values:

- *true:*
  Always generate properties to use JAXBElement, unless overriden by `<jaxb:property generateElementProperty="false"/>` on individual property.
- *false:*
  When generating properties from `<element nillable="true" minOccurs="0"/>`, generate a property not to use JAXBElement, as if the element declaration were just `<element nillable="true" />`, unless overriden by `<jaxb:property generateElementProperty="true"/>` on individual property. It is an error to specify this customization, when the property is required to be JAXBElement (such as when a property contains multiple elements with different names but of the same type.)

7.6 <schemaBindings> Declaration

The customization values in `<schemaBindings>` binding declaration have schema scope. This binding declaration is therefore useful for customizing at a schema level.

7.6.1 Usage

```
<schemaBindings [ map="boolean" ] >
    [ <package> package </package> ]
    [ <nameXmlTransform> ... </nameXmlTransform> ]*
</schemaBindings>

<package [ name = "packageName" ]
    [ <javadoc> ... </javadoc> ]
</package>

<nameXmlTransform>
    [ <typeName       [ suffix="suffix" ]
                      [ prefix="prefix" ] /> ]
    [ <elementName    [ suffix="suffix" ]
                      [ prefix="prefix" ] />]
    [ <modelGroupName [ suffix="suffix" ]
                      [ prefix="prefix" ] />]
    [ <anonymousTypeName [ suffix="suffix" ]
                         [ prefix="prefix" ] />]
</nameXmlTransform>
```

For readability, the <nameXmlTransform> and <package> elements are shown separately. However, they are local elements within the <schemaBindings> element.

The following customizations are defined in the schema scope:

- *map* if specified, prevents the classes from being generated from this schema. When the value is “0” or “false”, then no class/interface/enum will be generated from this package. map defaults to true.

The semantics of the customization value, if not specified above, are specified when they are actually used in the binding declarations.

For inline annotation, a <schemaBindings> is valid only in the annotation element of the <schema> element. There must only be a single instance of a <schemaBindings> declaration in the annotation element of the <schema> element.

If one source schema includes (via the include mechanism specified by XSD PART 1) a second source schema, then the <schemaBindings> declaration must be declared in the first including source schema. It should be noted that there is no such restriction on <schemaBindings> declarations when one

source schema imports another schema since the scope of <schemaBindings> binding declaration is schema scope.

7.6.1.1 package

Usage

- *name* if specified, defines the customization value *packageName*. *packageName* must be a valid Java package name.
- <*javadoc*> if specified, customizes the package level Javadoc. <*javadoc*> must be specified as described in Section 7.11, "<javadoc> Declaration." The Javadoc must be generated as specified in Section 7.11.3, "Javadoc Customization." The Javadoc section customized is the package section.

Design Note – The word "package" has been prefixed to *name* used in the binding declaration. This is because the attribute or element tag names "name" is not unique by itself across all scopes. For e.g., "name" attribute can be specified in the <property> declaration. The intent is to disambiguate by reference such as "packageName".

The semantics of the *packageName* is specified in the context where it is used. If neither *packageName* nor the <*javadoc*> element is specified, then the binding declaration has no effect.

Example: Customizing Package Name

```
<jaxb:schemaBindings>
    <jaxb:package name = "org.example.po" />
</jaxb:schemaBindings>
```

specifies "org.example.po" as the package to be associated with the schema.

7.6.1.2 nameXmlTransform

The use case for this declaration is the UDDI Version 2.0 schema. The UDDI Version 2.0 schema contains many declarations of the following nature:

```
<xs:element name="bindingTemplate" type="uddi:bindingTemplate"/>
```

The above declaration results in a name collision since both the element and type names are the same - although in different XML Schema symbol spaces. Normally, collisions are supposed to be resolved using customization. However, since there are many collisions for the UDDI V2.0 schema, this is not a convenient solution. Hence the binding declaration `nameXmlTransform` is being provided to automate name collision resolution.

The `nameXmlTransform` allows a *`suffix`* and a *`prefix`* to be specified on a per symbol space basis. The following symbol spaces are supported:

- `<typeName>` for the symbol space "type definitions"
- `<elementName>` for the symbol space "element definitions"
- `<modelGroupName>` for the symbol space "model group definitions."
- `<anonymousTypeName>` for customizing Java value class to which an anonymous type is bound.[1]

If *`suffix`* is specified, it must be appended to all the default XML names in the symbol space. The *`prefix`* if specified, must be prepended to the default XML name. Furthermore, this XML name transformation must be done after the XML name to Java Identifier algorithm is applied to map the XML name to a Java identifier. The XML name transformation must not be performed on customization names.

By using a different *`prefix`* and/or *`suffix`* for each symbol space, identical names in different symbol spaces can be transformed into non-colliding XML names.

`anonymousTypeName`

The `<anonymousTypeName>` declaration can be used to customize the suffix and prefix for the Java value class. If *`prefix`* is specified, then it must be prepended to the Java value class name for the anonymous type. If suffix is specified, it must be appended.

[1] XML schema does not associate anonymous types with a specific symbol space. However, `nameXmlTransform` is used since it provides a convenient way to customize the value class to which an anonymous type is bound.

7.7 <class> Declaration

This binding declaration can be used to customize the binding of a schema component to an element class, value class or interface/implementation class. The customizations can be used to specify:

- a name for the derived Java class.
- an alternative implementation of interface/implementation binding.

Specification of an alternate implementation for an interface allows implementations generated by a tool (e.g. based on UML) to be used in place of the default implementation generated by a JAXB provider.

The implementation class may have a dependency upon the runtime of the binding framework. Since a runtime was not specified for JAXB 1.0 interface/ implementation binding, the implementation class may not be portable across JAXB provider implementations. Hence one JAXB provider implementation is not required to support the implementation class from another JAXB provider.

7.7.1 Usage

```
<class [ name = "className" ]
       [ implClass= "implClass" ]>
       [ ref = "className" ]
       [ <javadoc> ... </javadoc> ]
</class>
```

- *`className`* is the name of the derived value class, if specified. It must be a legal Java class name and must not contain a package prefix. The package prefix is inherited from the current value of *package*.
- *`implClass`* if specified, is the name of the implementation class for *`className`* and must include the complete package name. Note that this customization only impacts the return value for `className`'s factory method. This customization is ignored when `new` is used to create instances of a schema-derived Value class.
- *`ref`* if specified, is the name of the value class that is provided outside the schema compiler. This customization causes a schema compiler to refer to this external class, as opposed to generate a definition. It must include the complete package name. This attribute is mutually exclusive with the *`className`* attribute and the *`implClass`* attribute.

- <javadoc> element, if specified customizes the Javadoc for the derived value class. <javadoc> must be specified as described in Section 7.11, "<javadoc> Declaration."

7.7.2 Customization Overrides

When binding a schema element's Java representation to a value class or a Java Element class, the following customization values override the defaults specified in Chapter 5. It is specified in a common section here and referenced from Section 7.7.3, "Customizable Schema Elements."

- **name:** The name is *className* if specified.
- **package name:** The name of the package is *packageName* inherited from a scope that covers this schema element.

  NOTE: The *packageName* is only set in the <package> declaration. The scope of *packageName* is schema scope and is thus inherited by all schema elements within the schema.
- **javadoc:** The Javadoc must be generated as specified in section Section 7.11.3, "Javadoc Customization." The Javadoc section customized is the class/interface section.

7.7.3 Customizable Schema Elements

7.7.3.1 Complex Type Definition

When <class> customization specified in the annotation element of the complex type definition, the complex type definition must be bound to a Java value class as specified in Section 6.3.2, "Java value class" applying the customization overrides as specified in Section 7.7.2, "Customization Overrides."

Example: Class Customization: Complex Type Definition To Java value class

XML Schema fragment:

```
<xs:complexType name="USAddress">
    <xs:annotation> <xs:appinfo>
        <jaxb:class name="MyAddress" />
    </xs:appinfo></xs:annotation>
    <xs:sequence>...</xs:sequence>
    <xs:attribute name="country" type="xs:string"/>
</xs:complexType>
```

Customized code:

```
// public class USAddress { // Default Code
public class MyAddress {  // Customized Code
    public String getCountry(){...}
    public void setCountry(String value){...}
    ...
}
```

7.7.3.2 Simple Type Definition

When `<class>` customization specified in the annotation element of a simple type definition, the simple type definition must be bound to a Java value class as specified in Section 6.2.2.2, “Bind to a JAXB mapped class” applying the customization overrides as specified in Section 7.7.2, “Customization Overrides.”

Example: Class Customization: Simple Type Definition To Java value class

XML Schema fragment:

```
<xs:simpleType name="SKU">
    <xs:annotation> <xs:appinfo>
        <jaxb:class/>
    </xs:appinfo></xs:annotation>
    <xs:restriction base=”xs:int”/>
</xs:simpleType>
```

Customized code:

```
public class SKU {
    @XmlValue
    public int getValue(){...}
    public void setValue(int value){...}
    ...
}
```

7.7.3.3 Model Group Definition

It is invalid to place a <jaxb:class> customization on a model group.

7.7.3.4 Model Group

It is invalid to place a <jaxb:class> customization on an unnamed model group.

7.7.3.5 Global Element Declaration

A <*class*> declaration is allowed in the annotation element of the global element declaration. However, the *implClass* attribute is not allowed. The global element declaration must be bound as specified in Section 6.7.2, "Bind to Element Class" applying the customization overrides specified in Section 7.7.2, "Customization Overrides."

Example: Class Customization: Global Element to Class

XML Schema Fragment:

```
<xs:complexType name="AComplexType">
    <xs:sequence>
        <xs:element name="A" type="xs:int"/>
        <xs:element name="B" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
<xs:element name="AnElement" type="AComplexType">
    <xs:annotation><xs:appinfo>
        <jaxb:class name="MyElement"/>
    </xs:appinfo></xs:annotation>
</xs:element>
```

Customized code:

```
// following class is generated because of customization
```

```
    public class AComplexType {
        void setA(int value) {...}
        int getA(){...}
        void setB(String value){...}
        String getB(){...}
    }
    public class MyElement extends JAXBElement<AComplexType> {...}
    public class ObjectFactory {
        // Default code
        // JAXBElement<AnElement> createAnElement(AnElement){...}

        // Customized code
        MyElement     createMyElement(AnElement){...}
        ... other factory methods ...
}
```

7.7.3.6 Local Element

A local element is a schema element that occurs within a complex type definition. A local element is one of:

- local element reference (using the "ref" attribute) to a global element declaration.
- local element declaration ("ref" attribute is not used).

A `<class>` declaration is allowed in the annotation element of a local element. Section 7.15, "Annotation Restrictions" contains more information regarding the annotation element for a local element reference. However, the *`implClass`* attribute is not allowed.

A `<class>` customization on local element reference must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations" since a local element reference is never bound to a Java Element class.

A `<class>` customization on local element declaration applies only when a local element declaration is bound to a Java Element class. Otherwise it must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations." If applicable, a local element must be bound as specified in Section 6.7.1, "Bind to `JAXBElement<T>` Instance" applying the customization overrides as specified in Section 7.7.2, "Customization Overrides."

Example: Class Customization: Local Element Declaration To Java Element

The following example is from Section 6.12.3.2, "Examples."

XML Schema fragment:

```
<xs:complexType name="Base">
    <xs:choice maxOccurs="unbounded">
        <xs:element name="A" type="xs:string">
            <xs:annotation><xs:appinfo>
                <jaxb:class name="Bar"/>
            </xs:appinfo></xs:annotation>
        </xs:element>
        <xs:element name="B" type="xs:string"/>
        <xs:element name="C" type="xs:int"/>
    </xs:choice>
</xs:complexType>
```

Customized code:

```
import javax.xml.bind.JAXBElement;
public class ObjectFactory {
    // element instance factories only
    // JAXBElement<String> createBaseA(String value); //default code
    JAXBElement<String> createBaseBar(String value); //Customized
    JAXBElement<String> createBaseB(String value);
    JAXBElement<Integer> createBaseC(Integer value);
}
public class Base {
    static public class Bar extends JAXBElement<String> {...}//
Customized code
    /**
     * A general content list that can contain element
     * instances of JAXBElement<String> or JAXBElement<Integer>.
     */
    List<Object> getBarOrBOrC(){...}
}
```

7.8 <property> Declaration

This binding declaration allows the customization of a binding of an XML schema element to its Java representation as a property. This section identifies all XML schema elements that can be bound to a Java property and how to customize that binding.

The scope of customization value can either be definition scope or component scope depending upon which XML schema element the *<property>* binding declaration is specified.

7.8.1 Usage

```
<property  [ name = "propertyName" ]
    [ collectionType = "propertyCollectionType" ]
    [ fixedAttributeAsConstantProperty= "true" | "false" | "1" | "0"
]
    [ generateIsSetMethod= "true" | "false" | "1" | "0" ]
    [ enableFailFastCheck="true" | "false" | "1" | "0" ]
    [ generateElementProperty= "true" | "false" | "1" | "0" ]
    [ attachmentRef = "resolve" | "doNotResolve" | "default" ]
    [ <baseType name="fully qualified Java class"> ... </baseType> ]
    [ <javadoc> ... </javadoc> ]
</property>

<baseType name="fully qualified Java class">
    <javaType> ... </javaType>
</baseType>
```

For readability, the `<baseType>` element is shown separately. However, it can be used only as a local element within the `<property>` element.

The use of this declaration is subject to the constraints specified in Section 7.8.2.3, "Usage Constraints."

The customization values defined are:

- *name* if specified , defines the customization value *propertyName;* it must be a legal Java identifier.
- *collectionType* if specified, defines the customization value *propertyCollectionType* which is the collection type for the

property. *propertyCollectionType* if specified, must be either "indexed" or any fully qualified class name that implements *java.util.List.*

- *fixedAttributeAsConstantProperty* if specified , defines the customization value *fixedAttributeAsConstantProperty*. The value must be one of "true", false", "1" or"0".
- *generateIsSetMethod* if specified, defines the customization value of *generateIsSetMethod.* The value must be one of "true", false", "1" or"0".
- *enableFailFastCheck* if specified, defines the customization value *enableFailFastCheck.* The value must be one of "true", false", "1" or"0".
- *@generateElementProperty* if specified, controls the generation of JAXBElement property. The value must be one of "true", "false", "1", or "0". The default is absence of the value. It is an error for this attribute to be present if this customization is attached to local or global attribute declarations. This customization affects the binding as follows. It is an error to specify this customization, when the property is required to be *JAXBElement* (such as when a property contains multiple elements with different names but of the same type.)
  - *true*: Always generate properties to use *JAXBElement*.
  - *false*: When generating properties from *<element nillable="true" minOccurs="0" />*, generate a property not to use JAXBElement, as if the element declaration were just *<element nillable="true"/>*.
- @attachmentRef has a default value of "default". This mode defers to default processing as specified in Section H.2.2, "Binding WS-I Attachment Profile ref:swaRef".

  When @attachmentRef value is *resolve* and the property's base type is or derives from xsd:anyURI, the schema-derived JAXB property has a base type of javax.activation.DataHandler and the property is annotated with @XmlAttachmentRef.

  Disabling autoresolving an element/attribute of type ref:swaRef: When @attachmentRef value is *doNotResolve* and the property's base type derives from standard schema type ref:swaRef, the schema-derived JAXB property has the base type String, derived from

`xsd:anyURI`, and `@XmlAttachmentRef` is not generated for the property.

- *<javadoc>* element, if specified customizes the Javadoc for the property's getter method. *<javadoc>* must be specified as described in Section 7.11, "<javadoc> Declaration."

7.8.2 baseType

The <baseType> element is intended to allow the customization of a base type for a JAXB property. This element can only be a child of <jaxb:property> element.

```
<baseType name="fully qualified Java class">
   <javaType> ... </javaType>
</baseType>
```

The `@name` attribute enables either the specialization or generalization of the default base type binding for a JAXB property. Child element <javaType> is used to convert the default base type to a Java class. These two mutual exclusive usages of the <baseType> customization are described below.

7.8.2.1 Conversion using Child element <javaType>

Optional child element *<javaType>*, if specified, defines the customization value *javaType* and must be specified as defined in Section 7.9, "<javaType> Declaration." The customization value defined has component scope. This customization converts the default base type's value for a simple type definition to the Java class specified by <javaType> name.

The schema-derived JAXB property is annotated with `@XmlJavaTypeAdapter` specified in Section 8. `@XmlJavaTypeAdapter.value()` is set to a generated class[2] that extends `javax.xml.bind.annotation.adapter.XmlAdapter`. The generated class' `unmarshal` method must call the <javaType> customization's parse method, which is specified in Section 7.9, "<javaType> Declaration. The generated class' `marshal` method must call the <javaType> customization's print method.

[2] There is no need to standardize the name of the generated class since `@XmlJavaTypeAdapter.value()` references the class.

7.8.2.2 Generalize/Specialize baseType with attribute @name

The name attribute for <baseType> enables more precise control over the actual base type for a JAXB property. This customization enables specifying an alternative base type than the property's default base type. The alternative base type must still be in the same class inheritance hierarchy as the default base type. The alternative base type must be either a super interface/class or subclass of the default Java base type for the property. The customization enables one to specialize or generalize the properties binding.

The name attribute value must be a fully qualified Java class name. When the default base type is a primitive type, consider the default Java base type to be the Java wrapper class of that primitive type.

Generalizing the basetype using this customization enables simple type substitution for a JAXB property representing with too restrictive of a default base type. To enable all possible valid type substitutions, the name attribute should be java.lang.Object. However, if for example, it is known that all type substitutions will share a more specific Java super interface/class than java.lang.Object, that Java class name can be used achieve a stronger typed binding. With this customization, the JAXB annotation generated for the property's @XmlElement.type() or @XmlAttribute.type() is still the default Java datatype for the element/attribute's schema-defined type.

The schema-derived customized JAXB property is annotated, either explicitly or by default mapping annotations, with the mapping annotation @XmlElement, specified in Section 8.10.1. The @XmlElement annotation element type is derived in terms of the abstract model properties for a element type definition summarized in Section F.1.4, "Element Declaration Schema Component," on page 349 as follows:

Table 7-1 Annotate JAXB property with @XmlElement element-value pairs

@XmlElement element	@XmlElement value
type	the java type binding of the element declaration's *{type definition}*

Note that the Java class for @XmlElement.type() can differ from the recommended JAXB property's base type to enable type substitution of java.lang.Object. This binding enables unmarshalling of the Element's simple content when it is not qualified with an xsi:type as the element's schema-

declared type. `@XmlElement.type()` acts as the default `xsi:type` for a JAXB property where the property's base type was generalized to allow for type substitution of an element declaration with a simple type definition.

Specializing the basetype using this customization generates stronger typing than default JAXB binding. For example, an XML element or attribute of `xs:IDREF` binds to `java.lang.Object` by default as specified in Section 6.8.2, "Binding an IDREF component to a Java property". If the schema only intends the reference to be to an element that binds to a specific type, the baseType @name schema customization can be used to specialize the binding.

CODE EXAMPLE 7-1 Specialize binding of an IDREF via customization

Given XML Schema fragment:

```
<xs:complexType name="Book">
    <xs:sequence>
        <xs:element name="author" type="xs:IDREF"/>
            <xs:annotation><xs:appinfo>
                <jaxb:property>
                    <jaxb:baseType name="AuthorBio.class"/>
                </jaxb:property>
            </xs:appinfo></xs:annotation>
        <!-- ... -->
    </xs:sequence>
</xs:complexType>
<xs:complexType name="AuthorBio">
    <xs:sequence><!-- ... --> </xs:sequence>
    <xs:attribute name="name" type="xs:ID"/>
</xs:complexType>
```

Schema-derived Java value class:

```
public class Book {
    @XmlIDREF
    AuthorBio getAuthor() {...}
    void setAuthor(AuthorBio referencedObj){...}
}
public class AuthorBio{
    @XmlID
    String getName(){...}
    void setName(String value){...}
}
```

7.8.2.3 Usage Constraints

The usage constraints on <property> are specified below. Any constraint violation must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations." The usage constraints are:

1. The <baseType> is only allowed with the following XML schema elements from the Section 7.8.4, "Customizable Schema Elements":
   a. Local Element, Section 7.8.4.4, "Local Element."
   b. Local Attribute, Section 7.8.4.2, "Local Attribute."
   c. ComplexType with simpleContent, Section 7.8.4.8, "ComplexType."
2. **<baseType> can either have a name attribute or a <javaType>, they both can not exist at the same time.**
3. **The *fixedAttributeAsConstantProperty* is only allowed with a local attribute, Section 7.8.4.2, "Local Attribute" , that is fixed.**
4. **If a <property> declaration is associated with the <complexType>, then a <property> customization cannot be specified on the following schema elements that are scoped to <complexType>:**
   a. Local Element
   b. Model group
   c. Model Group Reference

   The reason is that a <property> declaration associated with a complex type binds the content model of the complex type to a general content property. If a <property> declaration is associated with a schema element listed above, it would create a conflicting customization.

Design Note – A Local Attribute is excluded from the list above. The reason is that a local attribute is not part of the content model of a complex type. This allows a local attribute to be customized (using a <property> declaration) independently from the customization of a complex type's content model.

Example: Property Customization: simple type customization

```
<xs:complexType name="internationalPrice">
    ....
    <xs:attribute name="currency" type="xs:string">
        <xs:annotation> <xs:appinfo>
            <jaxb:property>
                <jaxb:baseType>
                    <jaxb:javaType name="java.math.BigDecimal"
    parseMethod="javax.xml.bind.DatatypeConverter.parseInteger"
    printMethod="javax.xml.bind.DatatypeConverter.printInteger"/>
                </jaxb:baseType>
            </jaxb:property>
        </xs:appinfo></xs:annotation>
    </xs:attribute>
</xs:complexType>
```

The code generated is:

```
public class InternationalPrice {
    // String getCurrency(); default
    java.math.BigDecimal getCurrency(){...} // customized
    public void setCurrency(java.math.BigDecimal val){...} //
customized
}
```

7.8.3 Customization Overrides

When binding a schema element's Java representation to a property, the following customization values override the defaults specified in Chapter 6. It is specified in a common section here and referenced from Section 7.8.4, "Customizable Schema Elements."

- **name:** If *propertyName* is defined, then it is the name obtained by mapping the name as specified in Section 7.5.2, "Customized Name Mapping."
- **base type:** The basetype is `propertyBaseType` if defined. The *propertyBaseType* is defined by a XML schema element in Section 7.8.4, "Customizable Schema Elements."
- **collection type:** The collection type is `propertyCollectionType` if specified; otherwise it is the `propertyCollectionType` inherited from a scope that covers this schema element.

- **javadoc:** The Javadoc must be generated as specified in section Section 7.11.3, "Javadoc Customization." The Javadoc section customized is the `method section.`
- If *`propertyBaseType`* is a Java primitive type and *`propertyCollectionType`* is a class that implements java.util.List, then the primitive type must be mapped to its wrapper class.

The following does not apply if local attribute is being bound to a constant property as specified in Section 7.8.4.2, "Local Attribute":

- If *`generateIsSetMethod`* is `"true"` or `"1",` then additional methods as specified in Section 5.5.4, "`isSet` Property Modifier" must be generated.
- If *`enableFailFastCheck`* is `"true"` or `"1"` then the type constraint checking when setting a property is enforced by the JAXB implementation. Support for this feature is optional for a JAXB implementation in this version of the specification.

7.8.4 Customizable Schema Elements

7.8.4.1 Global Attribute Declaration

A <*`property`*> declaration is allowed in the annotation element of the global attribute declaration.

The binding declaration does not bind the global attribute declaration to a property. Instead it defines customization values that have definition scope. The definition scope covers all local attributes (Section 7.8.4.2, "Local Attribute") that can reference this global attribute declaration. This is useful since it allows the customization to be done once when a global attribute is defined instead of at each local attribute that references the global attribute declaration.

7.8.4.2 Local Attribute

A local attribute is an attribute that occurs within an attribute group definition, model group definition or a complex type. A local attribute can either be a

- local attribute reference (using the "ref" attribute) to a global attribute declaration.

- local attribute declaration (“ref” attribute is not used).

A <*property*> declaration is allowed in the annotation element of a local attribute.Chapter 7, “Annotation Restrictions” contains more information regarding the annotation element for a local attribute reference. The customization values must be defined as specified in Section 7.8.1, “Usage” and have component scope.

If *javaType* is defined, then the *propertyBaseType* is defined to be Java datatype specified in the "name" attribute of the *javaType*.

- If *fixedAttributeAsConstantProperty* is “*true*" or “*1*" and the local attribute is a fixed, the local attribute must be bound to a Java Constant property as specified in Section 6.8.1, “Bind to a Java Constant property” applying customization overrides as specified in Section 7.8.3, “Customization Overrides.” The *generateIsSetMethod*, *choiceContentProperty* and *enableFailFastCheck* must be considered to have been set to false.
- Otherwise, it is bound to a Java property as specified in Section 6.8, “Attribute use” applying customization overrides as specified in Section 7.8.3, “Customization Overrides.”

Example: Customizing Java Constant Property

XML Schema fragment:

```
<xs:complexType name="USAddress">
    <xs:attribute name="country" type="xs:NMTOKEN" fixed="US">
        <xs:annotation><xs:appinfo>
            <jaxb:property name="MY_COUNTRY"
                           fixedAttributeAsConstantProperty="true"/>
        </xs:appinfo></xs:annotation>
    </xs:attribute>
</xs:complexType>
```

Customized derived code:

```
public class USAddress {
    public static final String MY_COUNTRY = "US"; // Customized Code
}
```

Example 2: Customizing to other Java Property

XML Schema fragment:

```
<xs:complexType name="USAddress">
    <xs:attribute name="country" type="xs:string">
        <xs:annotation><xs:appinfo>
            <jaxb:propertyname="MyCountry"/>
        </xs:appinfo></xs:annotation>
    </xs:attribute>
</xs:complexType>
```

Customized derived code:

```
public class USAddress {
    // public getString getCountry();     // Default Code
    // public void setCountry(string value);// Default Code
    public String getMyCountry(){...}       // Customized Code
    public void setMyCountry(String value){...} // Customized Code
}
```

Example 3: Generating IsSet Methods

XML Schema fragment:

```
<xs:attribute name="account" type = "xs:int">
    <xs:annotation><xs:appinfo>
        <jaxb:property generateIsSetMethod="true"/>
    </xs:appinfo></xs:annotation>
</xs:attribute>
```

Customized code:

```
public int getAccount();
public void setAccount(int account);
public boolean isSetAccount(); // Customized code
public void unsetAccount();    // Customized code
```

7.8.4.3 Global Element Declaration

A *<property>* declaration is allowed in the annotation element of a global element declaration. However, the usage is constrained as follows:

The binding declaration does not bind the global element declaration to a property. Instead it defines customization values that have definition scope. The definition scope covers all local elements (Section 7.8.4.4, “Local Element”) that can reference this global element declaration. This is useful since it allows the customization to be done once when a global element is defined instead of at each local element that references the global element declaration.

7.8.4.4 Local Element

A local element is a schema element that occurs within a complex type definition. A local element is one of:

- local element reference (using the “ref” attribute) to a global element declaration.
- local element declaration (“ref” attribute is not used).

A <property> declaration is allowed in the annotation element of a local element. Section 7.15, “Annotation Restrictions” contains more information regarding the annotation element for a local element reference.

The customization values must be defined as specified in Section 7.8.1, “Usage” and have component scope.

If *javaType* is defined, then the *propertyBaseType* is defined to be Java datatype specified in the "name" attribute of the *javaType*.

The local element must be bound as specified in Section 6.12.7, “Content Model Default Binding” applying customization overrides as specified in Section 7.8.3, “Customization Overrides.”

See example in “Example 3: Property Customization: Model Group To Content Property Set” in section Section 7.8.4.6, “Model Group.”

7.8.4.5 Wildcard

A <property> declaration is allowed in the annotation element of the wildcard schema component. The customization values must be defined as specified in Section 7.8.1, “Usage” and have component scope.

The wildcard schema component must be bound to a property as specified in Section 6.12.5, “Bind wildcard schema component” applying customization overrides as specified in Section 7.8.3, “Customization Overrides.”

Example: The following schema example is from UDDI V2.0

```
<xs:complexType name="businessEntityExt">
    <xs:sequence>
        <xs:any namespace="##other"
            processContents="strict"
            minOccurs="1" maxOccurs="unbounded">
            <xs:annotation><xs:appinfo>
                <jaxb:property name="Extension"/>
            </xs:appinfo></xs:annotation>
        </xs:any>
        ....
    </xs:sequence>
</xs:complexType>
```

Customized derived code:

```
public class BusinessEntityExt {
    ...
    // List getAny(); // Default Code
    List getExtension(){...} // Customized Code
```

```
}
```

7.8.4.6 Model Group

A <property> binding declaration is allowed in the annotation element of the compositor (i.e. <choice>, <sequence> or <all>). The customization values must be defined as specified in Section 7.8.1, "Usage" and have component scope.

The customized binding of a model group is determined by the following:

- choiceContentProperty attribute in <globalBindings>.

1. If *propertyBaseType* is defined and a <property> declaration is also present, then the customization overrides specified in Section 7.8.3, "Customization Overrides" must be applied by the model group's parent schema element to the property used to aggregate the Java value class.
2. **If *propertySet* is defined, then the model group's parent schema element must aggregate the property set as specified in Section 6.3.1.2, "Aggregation of Property Set."**

Example1: Property Customization: Model Group To ChoiceContent Property

XML Schema fragment

```
<xs:annotation><xs:appinfo>
    <jaxb:globalBindings choiceContentProperty="true"/>
</xs:appinfo></xs:annotation>
<xs:complexType name="AType">
    <xs:choice>
        <xs:element name="foo" type="xs:int"/>
        <xs:element name="bar" type="xs:string"/>
    </xs:choice>
</xs:complexType>
```

Customized derived code:

```
class ObjectFactory {
    JAXBElement<Integer> createAtypeFoo(Integer value);
    JAXBElement<String> createAtypeBar(String value);
```

```
}
public class AType {
    void setFooOrBar(Object o){...}        //customized code
    Object getFooOrBar(){...}              //customized code
}
```

The `choiceContentProperty` is required to bind the choice model group to a choice content property.

Example 2: Property Customization: Model Group To General Content Property

XML Schema fragment:

```
<xs:complexType name="Base">
    <xs:choice maxOccurs="unbounded">
        <xs:annotation><xs:appinfo>
            <jaxb:property name="items" />
        </xs:appinfo></xs:annotation>
        <xs:element name="A" type="xs:string"/>
        <xs:element name="B" type="xs:string"/>
        <xs:element name="C" type="xs:int"/>
    </xs:choice>
</xs:complexType>
```

Customized derived code:

```
public class Base {
    /**
     * A general content list that can contain
     * instances of Base.A, Base.B and Base.C.
     */
    // List getAOrBOrC(); - default
    List getItems(){...}// Customized Code
}
```

Example 3: Property Customization: Model Group To Content Property Set

XML Schema fragment:

```
<xs:complexType name="USAddress"/>
<xs:complexType name="PurchaseOrderType">
    <xs:sequence>
        <xs:choice>
            <xs:group   ref="shipAndBill"/>
            <xs:element name="singleUSAddress" type="USAddress">
                <xs:annotation><xs:appinfo>
                    <jaxb:property name="address"/>
                </xs:appinfo></xs:annotation>
            </xs:element>
        </xs:choice>
    </xs:sequence>
</xs:complexType>
<xs:group name="shipAndBill">
    <xs:sequence>
        <xs:element name="shipTo" type="USAddress">
            <xs:annotation><xs:appinfo>
                <jaxb:property name="shipAddress"/>
            </appinfo></annotation>
        </xs:element>
        <xs:element name="billTo" type="USAddress">
            <xs:annotation><xs:appinfo>
                <jaxb:property name="billAddress"/>
            </xs:appinfo></xs:annotation>
        </xs:element>
    </xs:sequence>
</xs:group>
```

Customized derived code:

```
public interface PurchaseOrderType {
    USAddress getShipAddress(); void setShipAddress(USAddress);
    USAddress getBillAddress(); void setBillAddress(USAddress);
    USAddress getAddress();     void setAddress(USAddress);
}
```

7.8.4.7 Model Group Reference

A model group reference is a reference to a model group using the “ref” attribute. A property customization is allowed on the annotation property of the

model group reference. Section Chapter 7, “Annotation Restrictions” contains more information regarding the annotation element for a model group reference.

The customization values must be defined as specified in Section 7.8.1, “Usage” and have component scope. A model group reference is bound to a Java property set or a list property as specified in Chapter 6, “Content Model Default Binding” applying customization overrides as specified in Section 7.8.3, “Customization Overrides.”

7.8.4.8 ComplexType

A <property> customization is allowed on the annotation element of a complex type. The customization values must be defined as specified in Section 7.8.1, “Usage” and have component scope. The result of this customization depends upon the content type of the complex type.

- If the content type of the content model is simple content, then the content model must be bound to a property as specified in Section 6.3.2.1, “Simple Content Binding.” applying the customization overrides as specified in Section 7.8.3, “Customization Overrides.” If *javaType* is defined, then the *propertyBaseType* is defined to be Java datatype specified in the "name" attribute of the *javaType*.
- For all other content types, the content model must be bound as specified in step 1. of Section 6.12.7, “Content Model Default Binding” applying the customization overrides as specified in Section 7.8.3, “Customization Overrides".

Design Note – The <property> declaration is not allowed on an annotation element of attribute group definition. However, attributes within the attribute group definition can themselves be customized as described in the “Local Attribute” section above. Section 7.8.4.2, “Local Attribute.”

7.9 <javaType> Declaration

A <javaType> declaration provides a way to customize the binding of an XML schema atomic datatype to a Java datatype, referred to as the *target Java datatype.* The target Java datatype can be a Java built-in data type or an

application specific Java datatype. This declaration also provides two additional methods: a *parse method* and a *print method*.

The parse method converts a lexical representation of the XML schema datatype into a value of the target Java datatype. The parse method is invoked by a JAXB provider's implementation during unmarshalling.

The print method converts a value of the target Java datatype into its lexical representation of the XML schema datatype. The print method is invoked by a JAXB provider's implementation during marshalling.

7.9.1 Usage

```
<javaType name="javaType"
            [ xmlType="xmlType" ]
            [ parseMethod="parseMethod" ]
            [ printMethod="printMethod" ]>
```

The binding declaration can be used in one of the following:

- a `<globalBindings>` declaration.
- annotation element of one of the XML schema elements specified in Section 7.9.6, "Customizable Schema Elements."
- in a `<property>` declaration. See Section 7.8, "`<property>` Declaration." This can be used for customization at the point of reference to a simple type.

When used in a `<globalBindings>` declaration, `<javaType>` defines customization values with global scope. When used in an annotation element of one of the schema elements specified in Section 7.9.6, "Customizable Schema Elements." the customization values have component scope.

7.9.1.1 name

The *`javaType`*, if specified, is the Java datatype to which *`xmlType`* is to be bound. Therefore, *`javaType`* must be a legal Java type name, which may include a package prefix. If the package prefix is not present, then the Java type name must be one of the Java built-in primitive types [JLS - Java Language Specification, Second Edition, Section 4.2, "Primitive Types and Values"]. (For example, "`int`") or a Java class in the unnamed package. If class javaType declares a public constructor with following signature,

javaType(java.lang.String),parseMethod attribute does not need to be specified.

7.9.1.2 xmlType

The *xmlType*, if specified, is the name of the XML Schema datatype to which *javaType* is to bound. If specified, *xmlType* must be a XML atomic datatype derived from restriction. The use of the *xmlType* is further constrained as follows.

The purpose of the *xmlType* attribute is to allow the global customization of a XML schema to Java datatype. Hence *xmlType* attribute is required when <javaType> declaration's parent is <globalBindings>. If absent, it must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations." Otherwise, the *xmlType* attribute must not be present since the XML datatype is determined from the XML schema element with which the annotation element containing <javaType> declaration or the <baseType> (containing the <javaType>) is associated. If present, it must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations."

Examples can be found in "Example: javaType Customization: Java Built-in Type" and "Example: javaType Customization: User Specified Parse Method"

7.9.1.3 parseMethod

The parse method if specified, must be applied during unmarshalling in order to convert a string from the input document into a value of the target Java datatype. The parse method must be invoked as follows:

- The parse method defaults to new provided *javaType* is not a Java primitive type such as ("int"). If *javaType* is a Java primitive type, then this must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations." Otherwise, the binding compiler must assume that the target type is a class that defines a constructor as follows:
  - String as the first parameter of the constructor.

  To apply the conversion to a string it must generate code that invokes this constructor, passing it the input string.
- The parse method may be specified in the form *ClassName.methodName,* where the *ClassName* is a fully qualified class name that includes the package name. A compiler must assume that the

class *ClassName* exists and that it defines a static method named *methodName* that takes:

- `String` as the first argument.

To apply the conversion to a string it must generate code that invokes this method, passing it the input string.

- The parse method may be specified in the form *methodName* provided *`javaType`* is not a Java primitive type (such as `"int"`). If *`javaType`* is Java primitive type, then this must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations." Otherwise, the binding compiler must assume that *methodName* is a method in the class *`javaType.`* The binding compiler must therefore prefix the *`javaType`* to the *methodName* and process *`javaType.`**methodName* as specified in above.

The string passed to parse method can be any lexical representation for `xmlType` as specified in [XSD PART2].

If parseMethod attribute is not specified, `xmlType` is not a primitive or wrapper class and `javaType` has an accessible one argument constructor, where the argument is type `java.lang.String`, input text is parsed by invoking `new` with a `java.lang.String` parameter.

7.9.1.4 printMethod

The print method if specified, must be applied during marshalling in order to convert a value of the target type into a lexical representation:

- The print method is specified in the form *methodName* provided *`javaType`* is not a Java primitive type (such as `"int"`). If *`javaType`* is Java primitive type, then this must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations." Otherwise, the compiler must assume that the target type is a class or an interface that defines a zero-argument instance method named *methodName* that returns a `String`. To apply the conversion it must generate code to invoke this method upon an instance of the target Java datatype.
- If the print method is specified in the form *ClassName.methodName* then the compiler must assume that the class *ClassName* exists and that it defines a static method named *methodName* that returns a string that takes the following:
  - the first parameter is the target Java datatype.

To apply the conversion to a string it must generate code that invokes this method, passing it a value of the target Java datatype.

The lexical representation to which the value of the target type is converted can be any lexical representation for `xmlType` as specified in [XSD PART2].

If `printMethod` attribute is not specified and `xmlType` is not a primitive or wrapper class, `javaType.toString()` is used as the default print method..

7.9.2 DatatypeConverter

Writing customized parse and print methods can be difficult for a Java programmer. This requires a programmer to understand the lexical representations of XML schema datatypes. To make it easier, an interface, `DatatypeConverterInterface`, and a class `DatatypeConverter` are defined to expose the parse and print methods of a JAXB implementation. These can be invoked by user defined parse and print methods. This shifts the burden of dealing with lexical spaces back to the JAXB implementation.

The `DatatypeConverterInterface` defines parse and print methods for XML schema datatypes. There is one parse and print method for each of XML schema datatype specified in Table 6-1, "Java Mapping for XML Schema Built-in Types," on page 77. The interface is fully specified by the Javadoc specified in `javax.xml.bind.DatatypeConverterInterface`.

The `DatatypeConverter` class defines a static parse and print method corresponding to each parse and print method respectively in the `DatatypeConverterInterface` interface. The property `javax.xml.bind.DatatypeConverter` can be used to select the name of a class that provides an implementation of the parse and print methods. The name specified in the property must be a fully qualified class name and must implement the interface `DatatypeConverterInterface`. The class is fully specified by the Javadoc specified in `javax.xml.bind.DatatypeConverter`.

7.9.2.1 Usage

The following example demonstrates the use of the `DatatypeConverter` class for writing a customized parse and print method.

Example: javaType Customization: User Specified Parse Method

This example shows the binding of XML schema type `"xs:date"` is bound to a Java datatype `long` using user specified print and parse methods.

```
<jaxb:globalBindings>
    <jaxb:javaType name="long" xmlType="xs:date"
                parseMethod="pkg.MyDatatypeConverter.myParseDate"
                printMethod="pkg.MyDatatypeConverter.myPrintDate"/>
    </jaxb:javaType>
</jaxb:globalBindings>

package pkg;
import javax.xml.bind.DatatypeConverter;
public class MyDatatypeConverter {
    public static long myParseDate(String s) {
        java.util.Calendar d = DatatypeConverter.parse(s);
        long result= cvtCalendarToLong(d) ; // user defined method
        return result;
    }
    public static String myPrintDate(long l) {
        java.util.Calendar d = cvtLongToCalendar(l); //user defined
        return DatatypeConverter.print(d);
    }
}
```

The implementation of the print methods (*`parseDate`* and *`printDate`*) are provided by the user.

The customization is applied during the processing of XML instance document. During unmarshalling, the JAXB implementation invokes *`myParseDate`*. If *`myParseDate`* method throws a *`ParseException`*, then the JAXB implementation code catches the exception, and generate a *`parseConversionEvent`*.

7.9.2.2 Lexical And Value Space

[XSD PART 2] specifies both a value space and a lexical space for an schema datatypes. There can be more than one lexical representation for a given value.

Examples of multiple lexical representations for a single value are:

- For boolean, the value `true` has two lexical representations `"true"` and `"1"`.

- For integer, the value `1` has two lexical representations `"1.0"` and `"1"`.

XSD PART 2 also specifies a canonical representation for all XML schema atomic datatypes.

The requirements on the parse and print methods are as follows:

- A JAXB implementation of a parse method in `DatatypeConverterInterface` must be capable of a processing all lexical representations for a value as specified by [XSD PART 2]. This ensures that an instance document containing a value in any lexical representation specified by [XSD PART 2] can be marshalled.
- A JAXB implementation of a print method in `DatatypeConverterInterface` must convert a value into any lexical representation of the XML schema datatype to which the parse method applies, as specified by [XSD PART 2] and which is valid with respect to the application's schema.

Design Note – The print methods that are exposed may not be portable. The only requirement on a print method is that it must output a lexical representation that is valid with respect to the schema. So two vendors can choose to output different lexical representations. However, there is value in exposing them despite being non portable. Without the print method, a user would have to be knowledgeable about how to output a lexical representation for a given schema datatype, which is not desirable.

7.9.3 Built-in Conversions

As a convenience to the user, this section specifies some built-in conversions. A built-in conversion is one where the parse and the print method may be omitted by a user. The built-in conversions leverage the narrowing and widening conversions defined in [JLS - Java Language Specification, Second Edition], Section 5.1.2, "Widening Primitive Conversion" and Section 5.1.3, "Narrowing Primitive Conversions." For example:

```
<xs:simpleType name="foo" type="xs:long">
   <xs:annotation><xs:appinfo>
      <jaxb:javaType name="int"/>
   </xs:appinfo></xs:annotation>
```

```
</xs:simpleType>
```

If the parse method is omitted, then a JAXB implementation must perform the one of the following binding options:

a. If *javaType* is one of the following primitive types or its corresponding wrapper class `byte, short, int, long, float, double`, bind *xmlType* to its default Java datatype using the parse method for the *xmlType* defined in `DatatypeConverter`. If necessary, convert the default Java datatype for `xmlType` to value of type *javaType* by a type cast.

b. Else if default Java datatype defines a public one-argument constructor that takes a `java.lang.String`, use `new` with a `java.lang.String` parameter for parsing.

c. Else javaType(java.lang.String) does not exist, this must result in an invalid binding customization as specified in Section 7.1.5, “Invalid Customizations.

Example: javaType Customization: Java Built-in Type

This example illustrates how to bind a XML schema type to a Java type different from the default one.

XML Schema fragment:

```
<xs:element name="partNumber" type="xs:int"/>
```

Customization:

```
<jaxb:globalBindings>
    ....
    <jaxb:javaTypename="long"
               xmlType="xs:int"/>
</jaxb:globalBindings>
```

Since a Java built-in is specified, a parse or a print method need not be specified. A JAXB implementation uses the parse and print methods defined in `DatatypeConverter` class for converting between lexical representations and values. A JAXB implementation unmarshals an input value using the following methods:

```
int j = (int)DataTypeConverter.parseLong(string);
```

7.9.4 Events

The parse method *parseMethod* may fail, since it is only defined on those strings that are valid representations of target Java datatype values and it can be applied to arbitrary strings. A parse method must indicate failure by throwing an exception of whatever type is appropriate, though it should never throw a `TypeConstraintException`. A JAXB unmarshaller process must ensure that an exception thrown by a parse method is caught and, if appropriate, a `parseConversionEvent` event is generated.

The print method *printMethod* usually does not fail. If it does, then the JAXB implementation must ensure that the exception thrown by a print method is caught and a `printConversionEvent` is generated.

7.9.5 Customization Overrides

The `<javaType>` overrides the default binding of *xmlType* to the Java datatype specified in Table 6-1, "Java Mapping for XML Schema Built-in Types," on page 77.

7.9.6 Customizable Schema Elements

7.9.6.1 Simple Type Definition

A `<javaType>` binding declaration is allowed in the annotation element of the of a simple type definition. The *javaType* overrides the default binding of *xmlType* to the Java datatype specified in Table 6-1, "Java Mapping for XML Schema Built-in Types," on page 77. The customization values defined have definition scope and thus covers all references to this simple type definition.

If the simple type definition is mapped to a schema-derived type, an `@XmlJavaTypeAdapter` is generated on that class. Annotation element `@XmlJavaTypeAdapter.value()` is set to a generated class[3] that extends `javax.xml.bind.annotation.adapter.XmlAdapter`. The generated class' `unmarshal` method must call the <javaType> customization's parse method, which is specified in Section 7.9, "`<javaType>` Declaration.

[3] There is no need to standardize the name of the generated class since `@XmlJavaTypeAdapter.value()` references the class.

The generated class' `marshal` method must call the <javaType> customization's print method.

7.9.6.2 GlobalBindings

A `<javaType>` binding declaration is allowed as part of `<globalBindings>`. The *`javaType`* overrides the default binding of *`xmlType`* to the Java datatype specified in Table 6-1, "Java Mapping for XML Schema Built-in Types," on page 77. The customization values defined have global scope.

For each element or attribute declaration that references an `xmlType` that has a globalBindings `<javaType>` customization specified for it, the corresponding JAXB property is annotated with `@XmlJavaTypeAdapter`.

7.9.6.3 <property><baseType> declaration

A `<javaType>` binding declaration is allowed as part of `<baseType>` in the `<property>` binding declaration. The *`javaType`* overrides the default binding of *`xmlType`* to the Java datatype specified in Table 6-1, "Java Mapping for XML Schema Built-in Types," on page 77. Additional semantics are specified in Section 7.8.2, "`baseType`" also apply.

The schema-derived JAXB property is annotated with `@XmlJavaTypeAdapter` as specified in Section 7.8.2, "`baseType`".

7.10 <typesafeEnum> Declaration

This binding declaration allows the customization of a binding of an XML schema element to its Java representation as an enum type, Section 8.9 in [JLS3]. Only simple type definitions with enumeration facets can be customized using this binding declaration.

7.10.1 Usage

```
<typesafeEnumClass>
    [ name = "enumClassName" ]
    [ map = "true" | "false" | "1" | "0" ]
    [ ref = "enumClassName" ]
    [ <typesafeEnumMember> ... </typesafeEnumMember> ]*
    [ <javadoc> enumClassJavadoc </javadoc> ]
</typesafeEnumClass>

<typesafeEnumMember name = "enumMemberName">
                  [ value = "enumMemberValue"]
    [ <javadoc> enumMemberJavadoc </javadoc> ]
</typesafeEnumMember>
```

There are two binding declarations *<typesafeEnumClass>* and *<typesafeEnumMember>*. The two binding declarations allow the enumeration members of an enumeration class and enumeration class itself to be customized independently.

The *<typesafeEnumClass>* declaration defines the following customization values:

- *name* defines the customization value *enumClassName*, if specified. *enumClassName* must be a legal Java Identifier; it must not have a package prefix.

  For an anonymous simple type, the `name` attribute must be present. If absent, it must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations."

- `map` determines if the simple type definition should be bound to an enum type. When `map`'s value is `false`, then the simple type definition must not be bound to an enum type. `map` defaults to `true`.

- `ref` if specified, is the name of the enum class that is provided outside the schema compiler. This customization causes a schema compiler to refer to this external enum, as opposed to generate a definition. It must include the complete package name. This attribute is mutually exclusive with the `className` attribute and the `map` attribute.

- *<javadoc>* element, if specified customizes the Javadoc for the enumeration class. *<javadoc>* defines the customization value *enumClassjavadoc* if specified as described in Section 7.11, "<javadoc> Declaration."

- Zero or more <typesafeEnumMember> declarations. The customization values are as defined as specified by the <typesafeEnumMember> declaration.

The <typesafeEnumMember> declaration defines the following customization values:

- *name* must always be specified and defines a customization value enumMemberName. enumMemberName must be a legal Java identifier.
- *value* defines a customization value enumMemberValue, if specified. enumMemberValue must be the enumeration value specified in the source schema. The usage of value is further constrained as specified in Section 7.10.2, "value Attribute."
- <javadoc> if specified, customizes the Javadoc for the enumeration constant. <javadoc> defines a customization value enumMemberjavadoc if specified as described in Section 7.11, "<javadoc> Declaration."

For inline annotation, the <typesafeEnumClass> must be specified in the annotation element of the <simpleType> element. The <typesafeEnumMember> must be specified in the annotation element of the enumeration member. This allows the enumeration member to be customized independently from the enumeration class.

7.10.2 value Attribute

The purpose of the *value* attribute is to support customization of an enumeration value using an external binding syntax. When the <typesafeEnumMember> is used in an inline annotation, the enumeration value being customized can be identified by the annotation element with which it is associated. However, when an external binding declaration is used, while possible, it is not desirable to use XPath to identify an enumeration value.

So when customizing using external binding syntax, the value attribute must be provided. This serves as a key to identify the enumeration value to which the <typesafeEnumMember> applies. It's use is therefore further constrained as follows:

- When <typesafeEnumMember> is specified in the annotation element of the enumeration member or when XPath refers directly to a

single enumeration facet, then the value attribute must be absent. If present, it must result in must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations."

- When <typesafeEnumMember> is scoped to the typesafeEnumClass declaration, the value attribute must be present. If absent, it must result in must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations." The *enumMemberValue* must be used to identify the enumeration member to which the <typesafeEnumMember> applies.

An example of external binding syntax can be found in "Example 2: typesafeEnum Customization: External Binding Declaration."

7.10.3 Inline Annotations

There are two ways to customize an enumeration class:

- split inline annotation
- combined inline annotation

In split inline annotation, the enumeration value and the enumeration class are customized separately i.e. the <typesafeEnumMember> is used independently not as a child element of <typesafeEnumClass>. An example of this is shown in "Example 1: typesafeEnum Customization: Split Inline Annotation."

In combined inline annotation, the enumeration value and the enumeration class are customized together i.e. the <typesafeEnumMember> is used as a child element of <typesafeEnumClass>. This is similar to the customization used in external binding declaration. In this case the value attribute must be present in the <typesafeEnumMember> for reasons noted in Section 7.10.2, "value Attribute." An example of this customization is shown in "Example 3: typesafeEnum Customization: Combined Inline Annotation."

7.10.4 Customization Overrides

When binding a schema type definition's Java representation to an enum type, the following customization values override the defaults specified in Chapter 5. It is specified in a common section here and referenced from Section 7.8.4, "Customizable Schema Elements."

- **name:** If *enumClassName* is defined, then the name obtained by mapping *enumClassName* as specified in Section 7.5.2, "Customized Name Mapping."
- **package name:** The name obtained by inheriting `packgeName` from a scope that covers this schema element and mapping *packageName* as specified in Section 7.5.2, "Customized Name Mapping."
- **enumclass javadoc:** `enumClassJavaDoc` if defined, customizes the `class/interface section (` Section 7.11.1, "Javadoc Sections") for the enumeration class, as specified in Section 7.11.3, "Javadoc Customization."
- **enum constant set:** Each member of the set is computed as follows:
  - **name:** If *enumMemberName* is defined, the name obtained by mapping *enumMemberName* as specified in Section 7.5.2, "Customized Name Mapping."
  - **javadoc:** `enumMemberJavaDoc` if defined, customizes the `field section (` Section 7.11.1, "Javadoc Sections") for the enumeration class, as specified in Section 7.11.3, "Javadoc Customization."

7.10.5 Customizable Schema Elements

Any XML Schema simple type which has an enumeration facet can be customized with `<jaxb:typesafeEnumClass>` declaration with the following exception. If the simple type definition derives from *`xs:QName. xs:NOTATIION, xs:base64Binary, xs:hexBinary, xs:date, xs:time, xs:dateTime, xs:duration, xs:gDay, xs:gMonth, xs:gYear, xs:gMonthDay, xs:gYearMonth, xs:IDREF, xs:ID`*, it must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations.". Since most of these Xml datatypes bind to a mutable Java type, instances of these Java types are not sufficient to be an immutable value of an enum constant.

Design Note – The rationale for not allowing a type definition that derives from `xs:ID` to bind to an enum type is to avoid complicating the resolution of `xs:IDREF` values to `xs:ID` values. It is easiest if `xs:ID` values are always mapped to an instance of `java.lang.String`.

Example 1: typesafeEnum Customization: Split Inline Annotation

XML Schema fragment:

```
<xs:simpleType name="USState">
    <xs:annotation><xs:appinfo>
        <jaxb:typesafeEnumClass name="USStateAbbr"/>
    </xs:appinfo></xs:annotation>
    <xs:restrictionbase="xs:NCName">
        <xs:enumeration value="AK">
            <xs:annotation><xs:appinfo>
                <jaxb:typesafeEnumMember name="STATE_AK"/>
            </xs:appinfo></xs:annotation>
        </xs:enumeration>
        <xs:enumeration value="AL">
            <xs:annotation><xs:appinfo>
                <jaxb:typesafeEnumMember name="STATE_AL"/>
            </xs:appinfo></xs:annotation>
        </xs:enumeration>
    </xs:restriction>
</xs:simpleType>
```

Customized derived code:

```
public enum USStateAbbr {
    STATE_AL, STATE_AK;
    public String value() { return name(); }
    public static USStateAbbr fromValue(String value) { ... }
};
```

Example 2: typesafeEnum Customization: External Binding Declaration

The following example shows how to customize the above XML schema fragment using an external binding syntax.

```
<jaxb:typesafeEnumClass name="USStateAbbr">
    <jaxb:typesafeEnumMember name="STATE_AK" value="AK">
    <jaxb:typesafeEnumMember name="STATE_AL" value="AL"/>
</jaxb:typesafeEnumClass>
```

The attribute `value` must be specified for `<typesafeEnumMember>`. This identifies the enumeration member to which `<typesafeEnumMember>` applies.

Example 3: typesafeEnum Customization: Combined Inline Annotation

The following example shows how to customize the above XML schema fragment using inline annotation which does not split the external binding syntax.

```
<xs:simpleType name="USState">
    <xs:annotation><xs:appinfo>
        <jaxb:typesafeEnumClass name="USStateAbbr">
            <jaxb:typesafeEnumMember name="STATE_AK" value="AK"/>
            <jaxb:typesafeEnumMember name="STATE_AL" value="AL"/>
        </jaxb:typesafeEnumClass>
    </xs:appinfo></xs:annotation>
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="AK"/>
        <xs:enumeration value="AL"/>
    </xs:restriction>
</xs:simpleType>
```

The attribute value must be specified for `typesafeEnumMember`. This identifies the enumeration member to which the binding declaration applies.

7.11 <javadoc> Declaration

The `<javadoc>` declaration allows the customization of a javadoc that is generated when an XML schema component is bound to its Java representation.

This binding declaration is not a global XML element. Hence it can only be used as a local element within the content model of another binding declaration. The binding declaration in which it is used determines the *section* of the Javadoc that is customized.

7.11.1 Javadoc Sections

The terminology used for the javadoc sections is derived from “Requirements for Writing Java API Specifications” which can be found online at http://java.sun.com/j2se/javadoc/writingapispecs/index.html.

The following sections are defined for the purposes for customization:

- package section (corresponds to package specification)
- class/interface section (corresponds to class/interface specification)
- method section (corresponds to method specification)
- field section (corresponds to field specification)

7.11.2 Usage

Note that the text content of a <javadoc> element must use CDATA or < to escape embedded HTML tags.

```
<javadoc>
    Contents in &lt;b>Javadoc&lt;\b> format.
</javadoc>
```

or

```
<javadoc>
    <<![CDATA[
    Contents in <b>Javadoc<\b> format
    ]]>
</javadoc>
```

7.11.3 Javadoc Customization

The Javadoc must be generated from the <javadoc> element if specified. The Javadoc section depends upon where <javadoc> element is used. JAXB providers may generate additional provider specific Javadoc information (for example, contents of the <xs:documentation> element).

7.12 <dom> Declaration

The <dom> customization binds an XML Schema component to DOM rather than to a strongly typed Java representation. Specifically, JAXB bindings for mixed content and wildcard result in a hybrid mixture of strongly typed Java instances with DOM nodes or java.lang.String, representing text info. These mixed bindings might be more easily processed solely as one form, namely as an XML fragment represented as DOM. This customization also meets a JAX-

WS 2.0 databinding requirement from Section 2.1.5.5, “Disabling Databinding”.

7.12.1 Usage

The syntax for the customization is the following:

```
<dom[ [type= "w3c" | otherDomRepresentations ] />
```

You can use the optional type attribute to specify the type of DOM. By default, it is W3C DOM.

7.12.2 Customizable Schema Elements

This customization can be attached to the following XML Schema components:

- Element declaration (`<xs:element>`)
- Type definition (`<xs:complexType>` and `<xs:simpleType>`)
- Wildcard (`<xs:any>`)
- Model groups (`<xs:choice>`,`<xs:all>`,`<xs:sequence>`)
- Model group definition (`<xs:group>`)
- Particle

For all of the above cases, the Java representation of the DOM element is an instance of the Element class for the specified DOM representation. For example, W3C DOM element is bound to `org.w3c.dom.Element`.

Special Case Handling of DOM customization on a:

- *type definition* - it is semantically equivalent to placing the dom customization on each element declaration referencing that type definition.
- *global element declaration* - it is semantically equivalent to placing the dom customization on each element declaration referencing, via `@ref`, the global element declaration. The dom customization on the global element declaration does not cause that element to be unmarshalled as DOM when it is the root element of an XML document nor when the element is part of a wildcard content JAXB property.
- *mixed content* - if an XML schema component is annotated with a `dom` customization and that XML schema component can contain character data information due to its parent complex type definition being defined

with mixed content, character data information is handled as specified in Section 6.12.4, “Bind mixed content”.

The dom customization allows one to disable databinding and process a part of a document using other technologies that require “raw” XML.

7.12.3 Examples

Wildcard Binding Example

A wildcard is mapped to a List of `org.w3c.dom.Element`. Each element that matches to the wildcard will be turned into a DOM tree.

```
<xs:complexType name=”foo”>
    <xs:sequence>
        <xs:any maxOccurs="unbounded" processContents="lax">
            <xs:annotation><xs:appinfo>
                <jaxb:dom/>
            </xs:appinfo></xs:annotation>
        </xs:any>
    </xs:sequence>
</xs:complexType>

import org.w3c.dom.Element;
public class Foo {
    @XmlAnyElement(lax=”false”)
    List<Element> getContent(){...}
}
```

Wildcard and Mixed Content Binding Example

If the complexType definition above is defined to have mixed content, due to element **`[complexType]`** having attribute `@mixed=”true”`, the JAXB binding is:

```
import org.w3c.dom.Element;
public class Foo {
    /* Element content is represented org.w3c.dom.Element.
     * Character data information is represented as instances of
     *  java.lang.String. */
    @XmlMixed
    @XmlAnyElement(lax=”false”)
    List<Object> getContent(){...}
}
```

7.13 <inlineBinaryData> Declaration

The <inlineBinaryData> customization provides declarative control over the optimization for binary data described in Appendix H, "Enhanced Binary Data Handling.

7.13.1 Usage

The syntax for the customization is the following:

```
<inlineBinaryData/>
```

This customization disables considering the binary data optimization for a schema component containing binary data.

This customization can be attached to the following XML Schema components:

- Element declaration (<xs:element>) with binary data or
- Type definition (<xs:complexType> and <xs:simpleType>) deriving from binary datatype

When a schema component that binds to a JAXB property is customized with <inlineBinaryData>, its schema-derived JAXB property is annotated with `@XmlInlineBinaryData`. When a type definition is customized with <inlineBinaryData>, its schema-derived class is annotated with program annotation `@XmlInlineBinaryData`.

7.14 <factoryMethod> Declaration

The <factoryMethod> customization provides declarative control over an element or type factory method name generated in a package's `ObjectFactory` class introduced in Section 5.2, "Java Package.". This customization is useful to resolve name collisions between factory methods in the schema-derived `ObjectFactory` class.

7.14.1 Usage

The syntax for the customization is the following:

```
<factoryMethod name="BaseForFactoryMethodName"/>
```

The customization value defined is:

- *name* – each character of name must be a valid part of a Java identifier as determined by `java.lang.Character.isJavaIdentifierPart()`.

The name of the factory method is generated by concatenating the following components:

- The string constant `create`
- `@name`'s value

7.14.1.1 Usage Constraints

The usage constraints on `<factoryMethod>` are specified below. Any constraint violation must result in an invalid customization as specified in Section 7.1.5, "Invalid Customizations." The usage constraints are:

1. `<factoryMethod>` is only allowed to annotate an element declaration or a type definition.

Note that this customization does not require a factory method to be generated, it simply provides a factory method name if a factory method is to be generated for the annotated element declaration or type definition. Section 6 and 7 specifies when a factory method is generated for an element declarations or type definitions.

7.15 Annotation Restrictions

[XSD PART 1] allows an annotation element to be specified for most elements but is ambiguous in some cases. The ambiguity and the way they are addressed are described here.

The source of ambiguity is related to the specification of an annotation element for a reference to a schema element using the "ref" attribute. This arises in three cases:

- A local attribute references a global attribute declaration using the "ref" attribute.

- A local element in a particle references a global element declaration using the “ref” attribute.
- A model group in a particle references a model group definition using the “ref” attribute.

For example in the following schema fragment (for brevity, the declaration of the global element “Name” and “Address” has been omitted).

```
<xs:element name = "Customer">
    <xs:complexType>
        <xs:element ref = "Name"/>
        <xs:element ref = "Address" />
    </xs:complexType>
</xs:element>
```

XML Schema spec is ambiguous on whether an annotation element can be specified at the reference to the “Name” element.

The restrictions on annotation elements has been submitted as an issue to the W3C Schema Working Group along with JAXB technology requirements (which is that annotations should be allowed anywhere). Pending a resolution, the semantics of annotation elements where the XML spec is unclear are assumed as specified as follows.

This specification assumes that an annotation element can be specified in each of the three cases outlined above. Furthermore, an annotation element is assumed to be associated with the abstract schema component as follows:

- The annotation element on an attribute ref is associated with {Attribute Use}
- The annotation element on a model group ref or an element reference is associated with the {particle}.

CHAPTER 8

JAVA TYPES TO XML

8.1 Introduction

This chapter specifies the mapping from program elements to XML Schema. The mapping includes a default as well as a customized mapping.

8.2 Overview

This section is non normative and provides a high level view of Java to XML Schema mapping targeted towards both JAXB application developers and JAXB implementation vendors.

8.2.1 Mapping Scope

The mapping covers program elements commonly used in the composition of a data model for an application: package, field, property and types (classes and enum construct). Additionally, the mapping scope also covers mapping annotations used to annotate schema derived code.

In so far as possible, a program element is mapped to an equivalent XML Schema construct in an intuitive manner. Thus,

- ***Package*** maps to a XML target namespace. A package provides a naming context for types. A XML target namespace provides a naming context for schema components such as elements, type definitions.
- ***Type*** maps to a schema type. A *value type* is a data container for values; e.g. a value class contains values represented by it's properties and

fields. A schema type is a datatype, an instance of which (e.g. element) acts as a data container for values represented by schema components within a schema type's content model (e.g. element, attributes, etc.). Thus a type maps naturally to a schema type. For e.g.,

- class typically maps to a complex type definition
- java primitive types and wrapper classes map to XML Schema simple type definition.

- ***Field or property*** maps to an element or an attribute contained within the complex type to which a type is mapped.
- ***Enum type*** maps to a simple schema type constrained by enumeration facets.

The input to the mapping process is one or more sets of packages or classes. A package or a class is mapped and recursively, fields, properties and types contained with it. The mapping is customizable.

8.2.2 Mapping Annotations

Mapping annotationsThe mapping of program elements to XML Schema construct can be customized using *mapping annotations*, program annotations based on JSR 175 program annotation facility. Mapping annotations are used for:

- customizing the Java to XML schema mapping.
- annotating schema derived code.
- control over optimized binary data encoding.

The mapping annotations are described in the `javax.xml.bind.annotation` and `javax.xml.bind.annotation.adapters` packages.

Retention Policy The retention policy of all mapping annotations is RetentionPolicy.RUNTIME. This policy allows introspection of mapping annotations at runtime. Introspection can be used by JAXB binding framework to marshal/unmarshal an object graph to XML representation or to customize the mapping of program elements to XML Schema constructs. This policy also allows a JAXB vendor implementation to generate a schema from a program element's compiled form rather than its source.

8.2.3 XML Name Derivation

Mapped program element is a program element that has been mapped to an XML Schema construct. It is possible to use `@XmlTransient` annotation type to prevent the mapping of a program element.

XML Names An XML name may be needed for the schema components for a mapped program element, for e.g. element name. XML names are usually derived from a program element name, for e.g. property, field, class name etc.But they can be customized using mapping annotation. When an XML name is derived from a property name, bean de capitalization rules are used. If a Java Identifier is not a legal XML name, then a legal XML name can be assigned using an annotation element (e.g. `@XmlType(name="foo")`).

8.2.4 Fields and Properties

XML global element Fields and properties typically map to local elements within a complex type for a class. But a well formed XML document begins with a root element (a global element in the corresponding schema). The `@XmlRootElement` annotation can be used to associate a global element with a class or an enum type.

Null Value and Nillable Element A null value for a type mapped to an XML Schema element in two ways: absence of an element or an nillable element. The mapping annotation for an element allows either mapping.

8.2.5 Type Mapping

Legacy applications One of the primary use cases for Java language to XML Schema mapping is to allow an existing application to be exported as a web service. In many cases, the existing applications are legacy applications consisting of classes that follow different class designs. The annotations and default mapping are designed to enable such classes to be mapped to schema with minimal changes to existing code. See Section 8.12, “Default Mapping,” on page 287 for default mapping.

Class A class usually maps to a complex type. However, using `@XmlValue` annotation, a class can also be mapped to a simple type (to hold a simple value) or a complexType with simpleContent (to hold a simple value and attributes).

The `@XmlType` annotation can be used to customize the mapping of a class. For example, it can be used to map a class to an anonymous type or to control the ordering of properties and/or fields. Properties and fields are unordered; but they can be mapped to a content model that is ordered (e.g. xs:sequence) or unordered content model (xs:all).

Class Designs A class with a public or protected no-arg constructor can be mapped. If a class has a ***static zero-arg factory*** method, then the factory method can be specified using the annotation element `@XmlType.factoryMethod()` and `@XmlType.factoryClass()`.

Ordering of Properties/fields: The ordering of properties and fields can be customized in one of two ways: at the package level using @XmlAccessorOrder or using `@XmlType.propOrder()` at the class level.

Class Hierarchy Mapping Class hierarchy typically maps to a type derivation hierarchy. The `@XmlType` and `@XmlValue` annotations together provide support mapping class hierarchy to schema type hierarchy where XML Schema complex type derives by extension from either another complex type or a simple type.

Supported Collection Types Typed collections and untyped collections are mapped. Mapped collection types are: arrays, indexed properties and parametric types. Mapped untyped collection are: `java.util.List`, `java.util.Set` and `java.util.HashMap`. Of these, `java.util.HashMap` does not map naturally to a XML Schema construct. For example, `HashMap` can have different XML serialized forms which differ in trade-offs made between memory and speed or specificity and generality. The XML serialization form can be customized using `@XmlJavaTypeAdapter` (Section 8.2.6, “Adapter”).

Collection serialized forms A collection type can be mapped to a XML Schema complex type and collection item is mapped to local element within it. Alternately, a parameterized collection (e.g. List<***Integer***>) can be mapped to a simple schema type that derives by list.

When a collection type is mapped to a XML Schema complex type, the mapping is designed to support two forms of serialization shown below.

```
//Example: code fragment
int[] names;

// XML Serialization Form 1 (Unwrapped collection)
// Element name is derived from property or field name
<names> ... </names>
<names> ... </names>
...

// XML Serialization Form 2 ( Wrapped collection )
// Element name of wrapper is derived from property or field name
// Element name of each item in collection is also derived from
// property name
<names>
    <names> value-of-item </names>
    <names> value-of-item </names>
    ....
</names>
```

The two serialized XML forms allow a null collection to be represented either by absence or presence of an element with a nillable attribute. The `@XmlElementWrapper` annotation on the property or field is used to customize the schema corresponding to the above XML serialization forms.

A parameterized collection (e.g. List<***Integer***>) can also be mapped to simple schema that derives by list using `@XmlList` annotation. For e.g. the serialized XML form is: “1 2 3 “.

8.2.6 Adapter

A type may not map naturally to a XML representation (see Supported Collection Types above). As another example, a single instance of a type may have different on wire XML serialization forms.

Adapter approach defines a portable customization mechanism for applications exemplified above. The mechanism provides a way to adapt a *bound type*, a Java type used to process XML content, to *value type*, mapped to an XML representation or vice versa. It is the value type that is used for marshalling and unmarshalling. Use of this approach involves two steps:

- provide an adapter class that extends the abstract class `@javax.xml.bind.annotation.adapters.XmlAdapter` that defines two methods `unmarshal()` and `marshal()`. The

methods are invoked by JAXB vendor implementation during unmarshalling and marshaling respectively to adapt between bound and value types.

- specify the adapter class using the `@XmlJavaTypeAdapter` annotation.

8.2.7 Referential Integrity

Preserving referential integrity of an object graph across XML serialization followed by a XML de serialization, requires an object reference to be marshalled by reference or containment appropriately. Possible strategies include:

- marshal all references to a given object by reference.
- marshal the first reference to an object by containment and subsequent references to the same object by reference.

Depending on the strategy, the schema to which program element is mapped also varies accordingly.

Two annotations `@XmlID` and `@XmlIDREF` provide the mechanism which can be used together to map program element by reference or containment. This places the burden of preserving referential integrity on a developer. On the other hand, the ability to customize the mapping is useful since it can result in mapping of program elements to a schema that defines a document structure more meaningfully to an application than a default derived schema.

8.2.8 Property/Field Name Collision

A XML name collision can arise when the property name obtained by bean de capitalization and the name of a field map to a same schema component. For example

```
public int item;
pubilc int getItem();
public void setItem(int val);
```

The name collision occurs because the property name, bean de capitalization, and the name of the public field are both the same i.e. `item`. In the case, where the property and the public field refer to the same field, the

`@XmlTransient` can be used to resolve the name collision by preventing the mapping of either the public field or the property.

8.3 Naming Conventions

Any source and schema fragments and examples shown in this chapter are meant to be illustrative rather than normative.

- `@XmlAttribute` denotes both a program annotation type as well a specific use of annotation type.
- The prefix `xs:` is used to refer to schema components in W3C XML Schema namespace.
- The prefix `ref:` is used to refer to schema components in the namespace "`http://ws-i.org/profiles/basic/1.1/xsd`"

Design Note – The mapping of program elements to schema components is specified using the abstract schema component model in XML Schema Part 1. The use of abstract schema components allows precise specification of the mapping and is targeted towards JAXB implementation vendors. In contrast, javax.xml.bind.annotation Javadoc is targeted towards the JAXB application developer. Hence it is the Javadoc that contains code and schema fragment samples.

Default mapping is specified in terms of customizations. First the mapping of program element to a schema component with the binding annotation is specified. Then the default mapping for a program element is specified by defining a default binding annotation. In the absence of any binding annotation, the default binding annotation is considered to annotate the program element.

For ease of reading, a synopsis of each program annotation is included inline in this chapter. Details can be found in the Javadoc published separately from this document.

8.4 Constraint Violations

For the purpose of mapping and constraint checking, if a program element is not annotated explicitly, and there is a default mapping annotation defined for that element, it must be applied first before performing any constraint checks or mapping. This is assumed in the normative mapping tables shown below.

The mapping of program elements to XML Schema constructs is subject to mapping constraints, specified elsewhere in this chapter. The mapping constraints must be enforced by the `javax.xml.bind.annotation.JAXBContext.newInstance(..)` method. Any cycles resulting from a combination of annotations or default mapping must be detected in `javax.xml.bind.annotation.JAXBContext.newInstance(..)` method and also constitutes a constraint violation. A `javax.xml.bind.JAXBException` or (its subclass, which can be provider specific) must be thrown upon a constraint violation.

A JAXB Provider must support the schema generation at runtime. See `javax.xml.bind.JAXBContext.generateSchema(..)` for more information.

8.5 Type Mapping

This section specifies the mapping of Java types to XML Schema.

8.5.1 Java Primitive types

The default mapping of Java types (and their wrapper classes) specified in table Table 8-1, “Mapping: Java Primitive types to Schema Types,” on page 226 must be supported.

Table 8-1 Mapping: Java Primitive types to Schema Types

Java Primitive Type	XML data type
`boolean`	`xs:boolean`
`byte`	`xs:byte`

Table 8-1 Mapping: Java Primitive types to Schema Types

Java Primitive Type	XML data type
short	xs:short
int	xs:int
long	xs:long
float	xs:float
double	xs:double

8.5.2 Java Standard Classes

The default mapping of Java classes specified in Table 8-2, “Mapping of Standard Java classes,” on page 228 must be supported.

Table 8-2 Mapping of Standard Java classes

Java Class	XML data type
java.lang.String	xs:string
java.math.BigInteger	xs:integer
java.math.BigDecimal	xs:decimal
java.util.Calendar	xs:dateTime
java.util.Date	xs:dateTime
javax.xml.namespace.QName	xs:QName
java.net.URI	xs:string
javax.xml.datatype.XMLGregorianCalendar	xs:anySimpleType
javax.xml.datatype.Duration	xs:duration
java.lang.Object	xs:anyType
java.awt.Image	xs:base64Binary
javax.activation.DataHandler	xs:base64Binary
javax.xml.transform.Source	xs:base64Binary
java.util.UUID	xs:string

Design Note – JAXP 1.3 package javax.xml.datatype introduced the following classes for supporting XML schema types: Duration and XMLGregorianCalendar. XMLGregorianCalendar supports for 8 schema calendar types - xs:date, xs:time, xs:dateTime, 6 g* types, all of which derive from xs:anySimpleType. The particular schema type is computed based on values of member fields of XMLGregorianCalendar. Since the actual schema type is not known until runtime, by default, XMLGregorianCalendar can only be mapped to xs:anySimpleType and an instance of XMLGregorianCalendar could be marshaled using xsi:type to specify the appropriate schema calendar type computed at runtime. However, the mapping can be customized

A byte[] must map to xs:base64Binary by default.

8.5.3 Generics

8.5.3.1 Type Variable

The following grammar is from [JLS], Section 4.4, “Type Variables”.

```
TypeParameter:
    TypeVariable TypeBound_opt

TypeBound:
     extends ClassOrInterfaceType AdditionalBoundListopt
```

A type variable without a *Typebound* must be mapped to xs:anyType.

A type variable with a *TypeBound* must map to the schema type to which *ClassOrInterfaceType* is mapped; the mapping of *ClassOrInterface* is subject to the mapping constraints specified in other sections in this chapter.

```
<!-- code fragment
public class Shape <T> {
    public T xshape;
    public Shape() {};
    public Shape(T f) {
    xshape = f;
    }
}

<!-- XML Schema
```

```
<xs:complexType name="shape">
    <xs:sequence>
      <xs:element name="xshape" type="xs:anyType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
```

8.5.3.2 Type Arguments and Wildcards

The following grammar is from [JLS], Section 4.5.1, “Type Arguments and Wildcards”.

```
TypeArguments:
    <ActualTypeArgumentList>

ActualTypeArgumentList:
    ActualTypeArgument
    ActualTypeArgumentList, ActualTypeArgument

ActualTypeArgument:
    ReferenceType
    Wildcard

Wildcard:
?WildcardBounds

WildcardBounds:
    extends ReferenceType
    super ReferenceType
```

A wildcard without a *WildcardBounds* must map to schema type xs:anyType.

A wildcard with a *WildcardBounds* whose super type is *ReferenceType* must map to schema type xs:anyType.

A wildcard with a *WildcardBounds* that extends a *ReferenceType* must map to the schema type to which the *ReferenceType* is mapped; this mapping is subject to the mapping constraints specified in other sections in this chapter and is determined by the annotations as specified in the mapping tables in the chapter. For example:

```
/** EXAMPLE : WildcarType Mapping
// Code fragment
public class Shape {...}

public class Rectangle extends Shape {...}
public class Circle extends Shape {...}

public class Foo {
    public java.util.List<? extends Shape> shapes;
}

<-- XML Schema fragment
<xs:complexType name="shape">
    ...
</xs:complexType>

<xs:complexType name="circle">
    <xs:complexContent>
        <xs:extension base="shape">
            ...
       </xs:extension>
     </xs:complexContent>
  </xs:complexType>

<xs:complexType name="rectangle">
    <xs:complexContent>
        <xs:extension base="shape">
            ...
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<xs:complexType name="foo">
    <xs:sequence>
        <xs:element name="shapes" type="shape" nillable="true"
                            maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
```

8.5.4 Collections

The following collection must be supported:

- `java.util.Map` and its subtypes (e.g. java.util.HashMap)
- `java.util.Collection` and it's subtypes (e.g. java.util.List)

The mapping of collection depends upon the annotations on the program elements and is specified in the mapping tables. This specification uses a *collection type* to be one of `java.util.Collection` (or a subtype derived from it) , an array or an JavaBean index property.

8.6 Java Package

`@XmlSchema` is used in the mapping of package to an XML target namespace.

8.6.1 @XmlSchema

8.6.1.1 Synopsis

```
public enum XmlNsForm {UNQUALIFIED, QUALIFIED, UNSET}

@Retention(RUNTIME) @Target({})
public @interface XmlNs {

@Retention(RUNTIME) @Target({PACKAGE})
public @interface XmlSchema {
    XmlNs[]  xmlns() default {};
    String namespace() default "";
    String location() default "";
    XmlNsForm elementFormDefault() default XmlNsForm.UNSET;
    XmlNsForm attributeFormDefault() default XmlNsForm.UNSET;
}
```

8.6.1.2 Mapping

If `location()` is "", a package annotated with `@XmlSchema` must be mapped as specified in Table 8-3, "Mapping: Package to XML target

namespace," on page 234. Otherwise a package will not produce any schema document.

Design Note – XML Schema Part 1 does not contain an abstract component definition for a schema. Neither is there a mapping of attribute information items (e.g. elementFormDefault) of the <schema> to properties of an abstract schema component. So the mapping below maps to attribute information items on the <schema> element. "absent" in the tables is used to mean absence of the attribute information item in the schema.

Design Note – When `location()` is present, this specification only guarantees that no schema is generated for the namespace. Implementations should generate `<import>` statements accordingly with the `schemaLocation` attribute pointing to the value of the `@XmlSchema.location()`, but `<import>` statements do not have corresponding schema components, and they are anyway just hints, so it's not possible to enforce such constraints. Implementations are also allowed to use values other than `@XmlSchema.location()` in `<import schemaLocation="..."/>` for example so that the reference points to a copy of the resource that's preferrable for the user.

Table 8-1 Mapping: Package to XML target namespace

`targetNamespace`	if `@XmlSchema.namespace()` is `""`, then `absent;` otherwise `@XmlSchema.namespace()`
`elementFormDefault`	if the value of `@XmlSchema.elementFormDefault()` is `@XmlNsForm.UNSET`, then `absent;` otherwise, the value of `@XmlSchema.elementFormDefault()`
`attributeFormDefault`	if the value of `@XmlSchema.attributeFormDefault()` is `@XmlNsForm.UNSET`, then `absent;` otherwise, the value of `@XmlSchema.attributeFormDefault()`
`Namespace prefixes`	if `@XmlSchema.xmlns()` is {} then implementation defined; otherwise `@XmlSchema.xmlns()`

8.6.2 @XmlAccessorType

This annotation allows control over default serialization of fields and properties.

8.6.2.1 Synopsis

```
@Inherited @Retention(RUNTIME) @Target({PACKAGE, TYPE})
public @interface XmlAccessorType {
    XmlAccessType value() default XmlAccessType.PUBLIC_MEMBER;
}

public enum XmlAccessType { NONE, PROPERTY, FIELD, PUBLIC_MEMBER}
```

8.6.2.2 Mapping

The following mapping constraints must be enforced:

This annotation can be used only with the following other annotations: `@XmlType, @XmlRootElement`, `@XmlAccessorOrder, @XmlSchema, @XmlSchemaType, @XmlSchemaTypes, @XmlJavaTypeAdapters.` It can also be used with the following annotations at the package level: `@ XmlJavaTypeAdapter`.

See Section 8.12, “Default Mapping”.

8.6.3 @XmlAccessorOrder

This annotation allows control over the default ordering of properties and fields that are mapped to XML elements. Properties and fields mapped to XML attributes are not impacted by this annotation since XML attributes are unordered.

8.6.3.1 Synopsis

```
@Inherited @Retention(RUNTIME) @Target({PACKAGE, TYPE})
public @interface XmlAccessorOrder {
    XmlAccessOrder value() default XmlAccessOrder.UNDEFINED;
}

public enum XmlAccessOrder { UNDEFINED, ALPHABETICAL}
```

8.6.3.2 Mapping

The following mapping constraints must be enforced:

1. This annotation can be used only with the following other annotations: `@XmlType, @XmlRootElement`, `@XmlAccessorType, @XmlSchema, @XmlSchemaType, @XmlSchemaTypes, @XmlJavaTypeAdapters.` It can also be used with the following annotations at the package level: `@ XmlJavaTypeAdapter`.

If the value of `@XmlAccessorOrder.value()` is `XmlAccessOrder.ALHPHABETICAL`, then the default ordering of fields/properties is lexicographic order as determined by `java.lang.String.CompareTo((String anotherString).`

If the `@XmlAccessorOrder.value()` is `XmlAccessOrder.UNDEFINED`, then the default ordering of fields/properties is unspecified.

8.6.4 @XmlSchemaType

This annotation allows a customized mapping to a XML Schema built in type. This is useful where a Java type can map to more than one schema built in types. An example is `XMLGregorianCalendar` which can represent one of the eight schema built-in types.

8.6.4.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD, PACKAGE})
public @interface XmlSchemaType {
    String name();
    String namespace() default "http://www.w3.org/2001/XMLSchema";
    Class type() default DEFAULT.class;
    static final class DEFAULT {}
}
```

8.6.4.2 Mapping

The following mapping constraints must be enforced:

- name() must be an atomic simple type schema type (or a type that derives from it) to which the type of the property or field can be mapped from XML Schema -> Java as specified in Section 6.2.2, “Atomic Datatype”. Example

  ```
  // @XmlSchemaType can specify any one of the eight calendar types
  // that map to XMLGregorianCalendar.
  @XmlSchemaType(name=”date”)
  XMLGregorianCalendar foo;
  ```

- If the annotation is used as a package level annotation or within `@XmlSchemaTypes`, value of `@XmlSchemaType.type()` must be specified and must be the Java type that is being customized.
- If the annotation is used on a field or a method, then value of type() must be DEFAULT.class.
- This annotation can only be used with the following other annotations: `@XmlElement, @XmlAttribute, @XmlJavaTypeAdapter, @XmlJavaTypeAdapters.`

package:

When this annotation is used at the package level, the mapping applies to references to `@XmlSchemaType.type()` as specified below. For clarity, the following code example is used along with normative text.

```
// Example: change the default mapping at package level
package foo;
@javax.xml.bind.annotation.XmlSchemaType
    (name="date",
    type=javax.xml.datatype.XMLGregorianCalendar.class)
```

A `@XmlSchemaType` that is specified as a package level annotation must apply at the point of reference as follows:

1. a property/field within a class in package (e.g `exmple.po)` whose reference type is `@XmlSchemaType.type()`. For e.g.

   ```
   // XMLGregorianCalendar will be mapped to XML Schema type "date"
   XMLGregorianCalendar cal;
   ```

2. a property/field within a class in package (e.g `exmple.po)`, where `@XmlSchemaType.type()` is used as a parametric type. For e.g.

   ```
   // Example: Following code maps to a repeating element with
   // XML Schema type of "date".
   List<XMLGregorianCalendar> bar;
   ```

***property/field*:**

A `@XmlSchemaType` specified on the property/field maps references to `@XmlSchemaType.type()` as follows:

1. property/field is a single valued.

   ```
   // Maps XMLGregorianCalendar to XML Schema type "date""
   @XmlSchemaType(name="date")
   public XMLGregorianCalendar cal;
   ```

2. a property/field where `@XmlSchemaType.type()` is used as a parametric type. For e.g.

   ```
   // Example: Following code maps to a repeating element with
   // XML Schema type of "date".
   @XmlSchemaType(name="date")
   List<XMLGregorianCalendar> bar;
   ```

8.6.5 @XmlSchemaTypes

This annotation is a container annotation for defining multiple `@XmlSchemaType` annotations at the package level.

8.6.5.1 Synopsis

```
@Retention(RUNTIME) @Target({PACKAGE})
public @interface XmlSchemaTypes {
    // Collection of @{@link XmlSchemaType} annotations
    XmlSchemaType[] value();
}
```

8.6.5.2 Mapping

Each `@XmlSchemaType` annotation in `@XmlSchemaTypes.value()` must be mapped as specified in Section 8.6.4, “@XmlSchemaType”.

8.7 Java class

8.7.1 @XmlType

`@XmlType` is used to map a Java class to a schema type. The schema type is computed from its annotation element values.

8.7.1.1 Synopsis

```
@Retention(RUNTIME) @Target({TYPE})
public @interface XmlType {
    String name() default "##default";
    String[] propOrder() default {""};
    String namespace() default "##default" ;
    Class factoryClass() default DEFAULT.class;
    static final class DEFAULT {};
    String factoryMethod() default "";
}
```

8.7.1.2 Mapping

The following mapping constraints must be enforced:

- a class must be either be a top level class or a nested static class.
- a class must have a public or protected no-arg constructor or a factory method identified by {`factoryClass(), factoryMethod()`} unless it is adapted using `@XmlJavaTypeAdapter`.
- If `factoryClass()` is other than DEFAULT.class, then `factoryMethod()` must be specified (i.e. the default value "" cannot be used.)
- If `factoryClass()` is `DEFAULT.class` and `factoryMethod()` is not "", then `factoryMethod()` be a method in this class.
- if `@XmlType.propOrder` is not {} or {""}, then the set must include all of the properties and fields mapped to particles as specified in:
  - Section 8.9.1, "@XmlElement"
  - Section 8.9.2, "@XmlElements"
  - Section 8.9.3, "@XmlElementRef"
  - Section 8.9.4, "@XmlElementRefs"
- `@XmlType.propOrder` must not include a field or property annotated with `@XmlTransient.`
- if the class, *subClass,* derives from another XML-bound class, *baseClass* directly or indirectly (other than java.lang.Object), then the *subClass* must not contain a mapped property or field annotated with `@XmlValue` annotation.
- If a class contains a mapped property or field annotated with `@XmlValue` annotation, then all other mapped fields or properties in the class must be mapped to an XML attribute.
- This annotation can be used with the following annotations: `@XmlRootElement, @XmlAccessorOrder, @XmlAccessorType.`
- Even though the syntax allows it, @XmlType is disallowed on an interface.

A class annotated with `@XmlType`, must be mapped as specified below:

- class must be mapped as specified in Table 8-6, “Mapping: Class to Simple Type Definition,” on page 243 if the class contains only one mapped property or field that is annotated with @XmlValue as specified in Section 8.9.10, “@XmlValue”.
- otherwise, the class must be mapped as specified in Table 8-4, “Mapping: Class to Complex Type Definition,” on page 240.

Table 8-1 Mapping: Class to Complex Type Definition

{name}	if @XmlType.name() is ““, then absent otherwise if @XmlType.name() is “##default“, then the XML name derived from the class name as specified in Section 8.12.1, “Java Identifier To XML Name” otherwise @XmlType.name()
{target namespace}	if @XmlType.namespace() is “##default“ && @XmlType.name() is ““ and class is annotated with @XmlRootElement, then the {target namespace} as specified in Table 8-7, “Mapping: Class to Element Declaration,” on page 246 otherwise if @XmlType.namespace() is “##default“ && @XmlType.name() is ““ and class is not annotated with @XmlRootElement, then the {target namespace} of the attribute or element to which the property or field, from where this class is referenced, is mapped. otherwise if @XmlType.namespace() is “##default“ && @XmlType.name() is not ““, then the namespace to which the package, in which class is defined, is mapped as specified in Table 8-3, “Mapping: Package to XML target namespace,” on page 234 otherwise @XmlType.namespace()

Table 8-1 Mapping: Class to Complex Type Definition

`{base type definition}`	if the class contains a mapped property or field annotated with `@XmlValue` as specified in Section 8.9.10, "@XmlValue", then the schema type to which mapped property or field's type is mapped. otherwise schema type to which the nearest XML-bound ancestor class is mapped **Note –** In the absence of an extends class, java.lang.Object is the implicit superclass of a class. java.lang.Object is by default bound to xs:anyType, the distinguished ur-type definition, the root of schema type definition hierarchy. In this case, the {derivation method} is mapped to restriction rather than by extension. java.lang.Object can be bound to xs:any using @XmlAnyElement. However, that annotation only applies to a property or field and thus has no impact on the mapping described here. **Note –** When class X with `@XmlType` derives from another class Y with `@XmlTransient`, which in turn derives from class Z with `@XmlType`, then the above wording causes complex type X to derive from complex type Z, causing Y to skip.
`{derivation method}`	if `{base type definition}` is `xs:anyType`, then by `restriction` otherwise `extension`
`{final}`	if class modifier `final` is present. then the set `{extension, restriction};` otherwise, the empty set.

Table 8-1 Mapping: Class to Complex Type Definition

`{abstract}`	`true` if the class modifier `abstract` is present; otherwise `false`.
`{attribute uses}`	The set of properties or fields mapped to attributes as specified in Section 8.9.7, "@XmlAttribute".
`{attribute wildcard}`	Attribute wildcard as specified in Section 8.9.8, "XmlAnyAttribute".
`{content type}`	1. *empty* if no mapped property or field is annotated with `@XmlElement` 2. `mixed` if a property or field is annotated with `@XmlMixed` as specified in Section 8.9.14, "@XmlMixed". 3. `simpleContent` if : a. no property or field is annotated with `@XmlElement` b. && one or more properties or fields is annotated with `@XmlAttribute` c. && one property is annotated with `@XmlValue`. 4. `element-only content` if one or more properties is annotated with `@XmlElement;` `content model` mapped as specified in Table 8-5, "Mapping: Class body to Model Group Component," on page 243".
`{prohibited substitutions}`	Empty set **Design Note –** This allows element declarations in content models to perform type substitution using the xsi:type attribute.
`{annotations}`	`absent`

Table 8-2 Mapping: Class body to Model Group Component

{compositor}	if @XmlType.propOrder() is {} then xs:all; otherwise xs:sequence. The ordering of particles is: if @XmlType.propOrder() is not "", then the order in which properties/fields are listed in @XmlType.propOrder(). if @XmlType.propOrder() is "" && class is annotated with @XmlAccessorOrder(XmlAcessOrder.ALPHABETICAL)) or @XmlAccessorOrder(XmlAccessOrder.ALPHABETICAL) is specified at the package level and class is not annotated with @XmlAccessorOrder(XmlAccessOrder.UNDEFINED), then alphabetical order as specified in Section 8.6.3, "@XmlAccessorOrder". otherwise order is unspecified.
{particles}	Set of properties or fields mapped to particles. See {compositor} mapping above for ordering of particles.
{annotation}	unspecified

Table 8-3 Mapping: Class to Simple Type Definition

{name}	if @XmlType.name() is "", then absent otherwise if @XmlType.name() is "##default", then the XML name derived from the class name as specified in Section 8.12.1, "Java Identifier To XML Name" otherwise @XmlType.name()

Table 8-3 Mapping: Class to Simple Type Definition

{target namespace}	if @XmlType.namespace() is "##default" && @XmlType.name() is "" and class is annotated with @XmlRootElement, then the {target namespace} as specified in Table 8-7, "Mapping: Class to Element Declaration," on page 246 otherwise if @XmlType.namespace() is "##default" && @XmlType.name() is "" and class is not annotated with @XmlRootElement, then the {target namespace} of the attribute or element to which the property or field, from where this class is referenced, is mapped. otherwise if @XmlType.namespace() is "##default" && @XmlType.name() is not "", then the namespace to which the package, in which class is defined, is mapped as specified in Table 8-3, "Mapping: Package to XML target namespace," on page 234 otherwise @XmlType.namespace()
{base type definition}	ur-type definition, xs:anyType. NOTE: This is subject to the mapping constraints on XmlType. See Section 8.7.1.2, "Mapping".
{facets}	empty set
{fundamental facets}	derived
{final}	empty set. A subset of {extension, list, restriction, union}.

Table 8-3 Mapping: Class to Simple Type Definition

{variety}	Must be mapped as shown below	
	atomic {primitive type definition}	if property or field type is one of: - primitive type - wrapper class - reference type mapped to a simple atomic type.
	list {item type definition}	if the property or field type is one of the following collection types: - generic list - indexed property - single dimensional array Section 8.13.1, "@XmlType: List simple type
	union {member type definitions}	Not mapped.
{annotation}	unspecified	

8.7.2 @XmlRootElement

`@XmlRooElement` can be used to associate a global element with the schema type to which a class is mapped.

8.7.2.1 Synopsis

```
@Retention(RUNTIME) @Target({TYPE})
public @interface XmlRootElement {
    String name() default "##default" ;
    String namespace() default "##default" ;
}
```

8.7.2.2 Mapping

The following mapping constraints must be enforced:

1. The only other annotations allowed with this annotation are: `@XmlType, @XmlEnum, @XmlAccessorType, @XmlAcessorOrder.`

A class annotated with `@XmlRootElement` annotation, must be mapped as specified in Table 8-7, “Mapping: Class to Element Declaration,” on page 246.

Table 8-1 Mapping: Class to Element Declaration

`{name}`	if `@XmlRootElement.name()` is “##default“, then the XML name derived from the class name as specified in Section 8.12.1, “Java Identifier To XML Name”; otherwise `@XmlRootElement.name()`
`{target namespace}`	if `@XmlRootElement.namespace()` is “`##default`“, then the value of the `targetNamespace` to which the package containing the class is mapped as specified in Table 8-3, “Mapping: Package to XML target namespace,” on page 234 otherwise `@XmlRootElement.namespace()`
`{type definition}`	schema type to which the class is mapped as specified in Section 8.7.1, “@XmlType”.
`{scope}`	`global`
`{value constraint}`	`absent`
`{nillable}`	`false`
`{identity-constraint definitions}`	empty set
`{substitution group affiliation}`	`absent` **Design Note –** The value is always absent since there is no mapping to a substitution group.

Table 8-1 Mapping: Class to Element Declaration

{substitution group exclusions}	{extension, restriction}
{disallowed substitution}	{substitution,extension,restriction}
{abstract}	false **Design Note –** A value of true indicates that the element is abstract and can occur in only content models when element has been substituted in a substitution group. Since there is no mapping to substitution groups, this value is always mapped to false.
{annotation}	unspecified

8.7.3 @XmlTransient

`@XmlTransient` is used to prevent the mapping of a class.

8.7.3.1 Synopsis

```
@Retention(RUNTIME) @Target(TYPE)
public @interface XmlTransient {}
```

8.7.3.2 Mapping

The class must not be mapped. Any reference to this class from the other XML-bound classes will treated as if they are refering to the nearest XML-bound ancestor of this class (which could be `java.lang.Object`, which guarantees that there always exists such a class.)

For the effect that this annotation causes on derived classes, see Table 8-4.

Note that a class with `@XmlTransient` may still have properties and fields with JAXB annotations. Those are mapped to XML when a derived class is mapped to XML. See section 8.9 for more details.

The following mapping constraints must be enforced:

- `@XmlTransient` is mutually exclusive with all other mapping annotations.

8.7.4 @XmlSeeAlso

`@XmlSeeAlso` is an annotation that can be optionally placed on a class to instruct the JAXB runtime and the schema generator to also bind classes listed in `@XmlSeeAlso`, when it binds the class that `@XmlSeeAlso` is on.

8.7.4.1 Synopsis

```
@Retention(RUNTIME) @Target(TYPE)
public @interface XmlRootElement {
    Class[] value();
}
```

8.8 Enum Type

8.8.1 @XmlEnum

8.8.1.1 Synopsis

```
@Retention(RUNTIME) @Target({TYPE}
public @interface XmlEnum {
    // Java type that is mapped to a XML simple type
    Class <?> value() default String.class;
}
```

8.8.1.2 Mapping

The following mapping constraints must be enforced:

- `@XmlEnum.value()` must be mapped to a XML schema simple type.

Table 8-1 Mapping: Enum type to Base Type Definition

`{base type definition}`	schema type to which `@XmlEnum.value()` is mapped.
`{variety}`	The value depends upon the schema type to which the `@XmlEnum.value()` is mapped. But syntactically, it is always a restriction of {base type definition}. derived from the {base type definition} **Note –** The {base type definition} may either be a list simple type or an atomic type. It will never be a union type because there is no mapping to union type for java->schema.

8.8.2 @XmlEnumValue

8.8.2.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD})
public @interface XmlEnumValue {
    String value();
}
```

8.8.2.2 Mapping

The following mapping constraints must be enforced:

- `@XmlEnumValue.value()` must have a valid lexical representation for `@XmlEnum.value()`.

Table 8-1 Mapping: Enum constant to Enumeration Schema Component

`{value}`	`@XmlEnumValue.value()`
`{annotation}`	unspecified

8.8.3 @XmlType

8.8.3.1 Synopsis

```
@Retention(RUNTIME) @Target({TYPE})
public @interface XmlType {
    String name() default "##default";
    String namespace() default "##default" ;
    String[] propOrder() default {""};
    Class factoryClass() default DEFAULT.class;
    static final class DEFAULT {};
    String factoryMethod() default "";
}
```

8.8.3.2 Mapping

The following mapping constraints must be enforced:

1. `factoryMethod()`, `factoryClass()` and `@XmlType.propOrder` must be ignored.
2. This annotation can be used only with the following other annotations: `@XmlRootElement`, `@XmlAccessorOrder`, `@XmlAccessorType`. However, `@XmlAccessorOrder` and `@XmlAccessorType` must be ignored; they are not meaningful when used to annotate an enum type.

Table 8-1 Mapping: Enum type to Simple Type Definition

`{name}`	if `@XmlType.name()` is "", then absent otherwise if `@XmlType.name()` is "`##default`", then the XML name derived from the enum type name as specified in Section 8.12.1, "Java Identifier To XML Name"; otherwise `@XmlType.name()`

Table 8-1 Mapping: Enum type to Simple Type Definition

{target namespace}	if @XmlType.namespace() is "##default" && @XmlType.name() is "" and enum type is annotated with @XmlRootElement, then the {target namespace} as specified in Table 8-11, "Mapping: Enum type to Element Declaration," on page 251 otherwise if @XmlType.namespace() is "##default" && @XmlType.name() is "" and enum type is not annotated with @XmlRootElement, then the {target namespace} of the attribute or element to which the property or field, from where this enum type is referenced, is mapped. otherwise if @XmlType.namespace() is "##default" && @XmlType.name() is not "", then the namespace to which the package, in which enum type is defined, is mapped as specified in Table 8-3, "Mapping: Package to XML target namespace," on page 234; otherwise @XmlType.namespace()
{base type definition}	Mapped as specified in Table 8-8, "Mapping: Enum type to Base Type Definition," on page 248.
{variety}	.Mapped as specified in Table 8-8, "Mapping: Enum type to Base Type Definition," on page 248.
{final}	{extension, restriction, list, union}
{facets}	the set constructed by mapping each enum constant to an enumeration schema component as specified in Table 8-9, "Mapping: Enum constant to Enumeration Schema Component," on page 248.
{fundamental facets}	empty set
{annotations}	unspecified

8.8.4 @XmlRootElement

@XmlRootElement can be used to associate a global element with the schema type to which the enum type is mapped.

The following mapping constraints must be enforced:

1. The only other annotations allowed with this annotation are: `@XmlType, @XmlEnum, @XmlAccessorType, @XmlAcessorOrder.` Note that `@XmlAccessorType` and `@XmlAccessorOrder` while allowed will be ignored by the constraint in Section 8.8.3.2, “Mapping,” on page 249.

The mapping must be performed as specified in Table 8-11, “Mapping: Enum type to Element Declaration,” on page 251.

Table 8-2 Mapping: Enum type to Element Declaration

`{name}`	if `@XmlRootElement.name()` is “##default“, then the XML name derived from the enum type name as specified in Section 8.12.1, “Java Identifier To XML Name”; otherwise `@XmlRootElement.name()`
`{target namespace}`	if `@XmlRootElement.namespace()` is “`##default`“, then the value of the `targetNamespace` to which the package containing the class is mapped as specified in Table 8-3, “Mapping: Package to XML target namespace,” on page 234 otherwise `@XmlRootElement.namespace()`
`{type definition}`	schema type to which the class is mapped as specified in Section 8.7.1, “@XmlType”.
`{scope}`	`global`
`{value constraint}`	`absent`
`{nillable}`	`false`
`{identity-constraint definitions}`	empty set
`{substitution group affiliation}`	`absent` **Design Note –** The value is always absent since there is no mapping to a substitution group.

Table 8-2 Mapping: Enum type to Element Declaration

{substitution group exclusions}	{extension, restriction}
{disallowed substitution}	{substitution,extension,restriction}
{abstract}	false
{annotation}	unspecified

8.9 Property And Field

The following must be mapped (subject to the mapping constraints listed below):

- read/write property as identified by `java.beans.Introspector.getBeanInfo` with its nearest XML-bound superclass as the stopClass.
- non static, non transient field of all the ancestors up to the stopClass (but excluding the stopClass itself); if annotated with `@XmlAttribute`, then static final field must be mapped (informally this maps to a fixed attribute but this is formally specified in the mapping tables below).

A *mapped property* is a property found as above and mapped either by default or using a JAXB annotation.

A *mapped field* is a field found as above and mapped either by default or using a JAXB annotation.

A property or field that has been annotated with `@XmlTransient` is not mapped.

The following mapping constraints must be enforced.

- For a property, a given annotation can be applied to either read or write property but not both.
- A property name must be different from any other property name in any of the super classes of the class being mapped.

- A mapped field name or the de capitalized name of a mapped property must be unique within a class. For e.g.

```
// Example 1:
// Both the field "x" and property getX/setX are mapped by
// default. However, the decapitalized name property getX/setX
// is also "x" which collides with the field name "x".
public class Foo {
    public int x;

    public int getX {..};
    public void setX {..};
}
```

8.9.1 @XmlElement

8.9.1.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD})
public @interface XmlElement {
    String name() default "##default" ; // name for XML element
    boolean nillable() default false;
    boolean required() default false;
    String namespace() default "##default" ;
    Class type() default DEFAULT.class;
    String defaultValue() default "\u0000";
    static final class DEFAULT {}
}
```

8.9.1.2 Mapping

The following mapping constraints must be enforced:

- The only additional mapping annotations allowed with `@XmlElement` are: `@XmlID`, `@XmlIDREF`, `@XmlList`, `@XmlSchemaType`, `@XmlValue`, `@XmlAttachmentRef`, `@XmlMimeType`, `@XmlInlineBinaryData`, `@XmlJavaTypeAdapter` and `@XmElementWrapper`. `@XmlElement` can also be used within `@XmlElements`.
- If the property or field type is a parametric collection type, then `@XmlElement.type()` must be `DEFAULT.class` or

`collectionitem.class` (since the type of the collection is already known).

A field or property annotated must be mapped as follows:

- If `@XmlElement.namespace()` is not "`##default`" and different from the `{target namespace}` of the enclosing class, then it must be mapped as specified in Section Table 8-12, "Mapping: Property/field to Particle - ref attribute," on page 255.
- If property is single valued, and it's type is annotated with `@XmlRootElement` and `@XmlType.name()= ""`, then the property must be mapped as specified in Section Table 8-12, "Mapping: Property/field to Particle - ref attribute," on page 255.

  Design Note: This mapping is designed to eliminate an infinite recursion. For example:

```
// Code fragment
@XmlRootElement
@XmlType(name="")
class Foo {
    Foo foo;
}
```

  In the absence of the above mapping, the above code would map to:

```
<schema>
    <element name="foo">
        <complexType>
            <sequence>
                <element name="foo" minOccurs="0">
                    <complexType>
                        ... infinite recursion ...
```

  With the above mapping, the code fragment would instead map to:

```
    <schema>
        <element name="foo">
            <complexType>
                <sequence>
                    <element ref="foo" minOccurs="0">
```

- otherwise, it must be mapped as Table 8-13, "Mapping: Property/field to Particle - no ref attribute," on page 255.

Design Note – A local element corresponds to two abstract schema components - a particle and an element declaration. This is reflected in the mapping shown below.

Table 8-1 Mapping: Property/field to Particle - ref attribute

`{min occurs}`	if `@XmlElement.required()` is true, then 1 if the property type is a primitive type or a multi dimensional array with a primitive type then 1 otherwise 0
`{max occurs}`	if the type of the property/field is not a collection type, then 1 otherwise `unbounded`.

`{term}`	element declaration as specified in Table 8-14, "Mapping: Property/field to Element declaration," on page 257 with the following overrides for the abstract schema component properties:
	`{scope}` is `global`
	`{value constraint}` is `absent`
	`{type definition}` is `xs:anyType` if the mapping results in two or more element decalarations with the same name.
	***Note*:** The above make the element a global element declaration rather than a local element declaration.

Table 8-2 Mapping: Property/field to Particle - no ref attribute

`{min occurs}`	if `@XmlElement.required()` is true, then 1 otherwise if the property type is a primitive type or a multi dimensional array with a primitive type then 1 otherwise 0

Table 8-2 Mapping: Property/field to Particle - no ref attribute

`{max occurs}`	if the type of the property/field is not a collection type, then 1; otherwise `unbounded`.
`{term}`	must be mapped as specified in Table 8-14, “Mapping: Property/field to Element declaration,” on page 257.

Table 8-3 Mapping: Property/field to Element declaration

{name}	if @XmlElement.name() is "##default", then the XML name derived from the property or field name as specified in Section 8.12.1, "Java Identifier To XML Name"; otherwise @XmlElement.name()
{target namespace}	if @XmlElement.namespace() is "##default ", then if the enclosing package has @XmlSchema annotation and is @XmlSchema.elementFormDefault is @XmlNsForm.QUALIFIED, then the namespace of the enclosing class. otherwise "" (which produces unqualified element in the default namespace). otherwise, @XmlElement.namespace()

Table 8-3 Mapping: Property/field to Element declaration

{type definition}	***Note*: *The order of type inference below is significant*.** if @XmlElement.type() is not DEFAULT.class , then the schema type to which @XmlElement.type() is mapped. otherwise if annotated with @XmlList, schema type derived by mapping as specified in Section 8.9.13, "@XmlList" otherwise if annotated with @XmlValue, schema type derived by mapping as specified in Section 8.9.10, "@XmlValue" otherwise if annotated with @XmlID, the schema type derived by mapping as specified in Section 8.9.11, "@XmlID" otherwise if annotated with @XmlIDREF, the schema type derived by mapping as specified in Section 8.9.12, "@XmlIDREF" otherwise if the property or field is a collection type, then the schema type derived by mapping the collection item type. otherwise the schema type to which the type of the property is mapped.
{scope}	complex type to which the property's or the field's containing class is mapped as specified in Section 8.6.1, "@XmlSchema".
{value constraint}	if @XmlElement.defaultValue() is "\u0000" then absent otherwise default value with the value @XmlElement.defaultvalue().
{nillable}	@XmlElement.nillable()
{identity-constraint definitions}	absent

Table 8-3 Mapping: Property/field to Element declaration

{substitution group affiliation}	absent
{substitution group exclusions}	{extension, restriction}
{disallowed substitution}	{extension, restriction,substitution}
{abstract}	false
{annotation}	unspecified

.

8.9.2 @XmlElements

8.9.2.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD,METHOD})
public @interface XmlElements {
    XmlElement[] value(); // collection of @XmlElement annotations
}
```

8.9.2.2 Mapping

The following mapping constraints must be enforced:

- If the property or field type is a parameterized collection type, then the size of the `@XmlElements.value()` must be 1.
- This annotation can be used only with the following annotations: `@XmlIDREF, @XmlElementWrapper, @XmlJavaTypeAdapter.`
- If `@XmlIDREF` is specified, then each `@XmlElement.type()` must contain a JavaBean property/field annotated with `@XmlID`.

The property or field must be mapped as follows:

- If the size of `@XmlElements.value()` is 1, then the property must be mapped as specified in Section 8.9.1, “@XmlElement”.

- otherwise it must be mapped as specified in Table 8-15, “Mapping: List of types to choice particle,” on page 260.

Table 8-1 Mapping: List of types to choice particle

{min occurs}	0
{max occurs}	unbounded
{term}	If {particples} row in Table 8-16, “Mapping: List of types to choice model group of elements,” on page 260 results in a single particle, then that single particle. Otherwise mapped as specified in Table 8-16, “Mapping: List of types to choice model group of elements,” on page 260

Table 8-2 Mapping: List of types to choice model group of elements

{compositor}	choice
{particles}	set obtained by mapping each @XmlElement in @XmlElements.value() as specified in Table 8-14, “Mapping: Property/field to Element declaration,” on page 257.
{annotation}	unspecified

8.9.3 @XmlElementRef

8.9.3.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD})
public @interface XmlElementRef {
    String name() default "##default" ; // name for XML element
    String namespace() default "##default" ;
    Class type() default DEFAULT.class;
    static final class DEFAULT {}
}
```

8.9.3.2 Mapping

The following mapping constraints must be enforced:

- The only other additional JAXB mapping annotations allowed with `@XmlElementRef` are: `@XmlElementWrapper` and `@XmlJavaTypeAdapter`.
- If the collection item type or property type (for single valued property) is `javax.xml.bind.JAXBElement`, then {`@XmlElementRef.name()`,`@XmlElementRef.namespace()`} must point an element factory method with an `@XmlElementDecl` annotation in a class annotated with `@XmlRegistry` (usually `ObjectFactory` class generated by the schema compiler) :
  a. `@XmlElementDecl.name()` must equal `@XmlElementRef.name()`
  b. `@XmlElementDecl.namespace()` must equal `@XmlElementRef.namespace()`.
- If the collection item type (for collection property) or property type (for single valued property) is not `javax.xml.bind.JAXBElement`, then the type referenced by the property or field must be annotated with `@XmlRootElement`.

A field or property annotated with the `@XmlElementRef` annotation must be mapped as follows:

- if the type of the property or field is single valued property, then it must be mapped as specified in Table 8-17, “Mapping: Property/field (property type single valued) to Particle with ref attribute,” on page 262
- otherwise (the type of the property or field is a parametric type), then it must be mapped as specified in Table 8-19, “Mapping: Property/Field (parametric type) to choice particle,” on page 263.

Table 8-1 Mapping: Property/field (property type single valued) to Particle with ref attribute

{min occurs}	1
{max occurs}	1
{term}	must be mapped as specified in Table 8-18, “Mapping: Property/field to Element declaration with ref attribute,” on page 262.

Table 8-2 Mapping: Property/field to Element declaration with ref attribute

{name}	if `@XmlElementRef.type()` is `@XmlElementRef.DEFAULT.class` and the property type is not `javax.xml.bind.JAXBElement`, then the XML name `@XmlRootElement.name()` on the type being referenced. otherwise if `@XmlElementRef.type()` is `@XmlElementRef.DEFAULT.class` and the parametric type or the property type (for single valued property) is a `javax.xml.bind.JAXBElement`, then the `@XmlElementRef.name()`

Table 8-2 Mapping: Property/field to Element declaration with ref attribute

{target namespace}	if @XmlElementRef.type() is @XmlElementRef.DEFAULT.class and the property type is not javax.xml.bind.JAXBElement, then the XML namespace of the type being referenced. otherwise if @XmlElementRef.type() is @XmlElementRef.DEFAULT.class and the property type is single valued and is javax.xml.bind.JAXBElement, then the @XmlElementRef.namespace()
{annotation}	unspecified

Table 8-3 Mapping: Property/Field (parametric type) to choice particle

{min occurs}	0
{max occurs}	unbounded
{term}	If {particles} row in Table 8-20, "Mapping: Property/field (parametric type) to choice model group of element refs," on page 263 results in single particle, then that single particle. Otherwise mapped as specified in Table 8-20, "Mapping: Property/field (parametric type) to choice model group of element refs," on page 263

Table 8-4 Mapping: Property/field (parametric type) to choice model group of element refs

{compositor}	choice
{particles}	set obtained by visiting parametric type and each of its derived types and if annotated with @XmlRootElement, then mapping the @XmlRootElement as specified in as specified in Table 8-18, "Mapping: Property/field to Element declaration with ref attribute," on page 262.
{annotation}	unspecified

8.9.4 @XmlElementRefs

8.9.4.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD,METHOD})
public @interface XmlElementRefs {
    XmlElementRef[] value();
}
```

8.9.4.2 Mapping

The following mapping constraints must be enforced:

- The only other additional JAXB mapping annotations allowed with @XmlElementRefs are: `@XmlElementWrapper` and `@XmlJavaTypeAdapter`.

The property or field must be mapped as specified in Table 8-21, “Mapping: List of element instances to choice particle,” on page 264.

Table 8-1 Mapping: List of element instances to choice particle

`{min occurs}`	`0`
`{max occurs}`	`unbounded`
`{term}`	If the {particles} row in Table 8-22, “Mapping: List of element instances to choice model group of element refs,” on page 264 results in a single particle, then that single particle. Otherwise mapped as specified in Table 8-22, “Mapping: List of element instances to choice model group of element refs,” on page 264

Table 8-2 Mapping: List of element instances to choice model group of element refs

`{compositor}`	`choice`

Table 8-2 Mapping: List of element instances to choice model group of element refs

{particles}	set obtained by mapping • each @XmlElementRef in @XmlElementRefs.value() as specified in Section 8.9.3, "@XmlElementRef". • if property is annotated with @XmlAnyElement, then the particle obtained by mapping as specified in Section 8.9.6, "@XmlAnyElement"
{annotation}	unspecified

8.9.5 @XmlElementWrapper

8.9.5.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD})
public @interface XmlElementWrapper {
    String name() default "##default" ; // name for XML element
    String namespace() default "##default" ;
    boolean nillable() default false;
    boolean required() default false;
}
```

8.9.5.2 Mapping

The following mapping constraints must be enforced:

- The only additional mapping annotations allowed with @XmlElementWrapper are: @XmlElement, @XmlElements, @XmlElementRef, @XmlElementRefs, @XmlJavaTypeAdapter.
- The property or the field must be a collection property.

The property or field must be mapped as follows:

- If @XmlElementWrapper.namespace() is not "##default" and different from the {target namespace} of the enclosing class, then it must

be mapped as specified as specified in Section Table 8-25, “Mapping: Property/field Element Wrapper with ref attribute,” on page 267.

- otherwise, it must be mapped as Table 8-23, “Mapping: Property/field to Particle for Element Wrapper,” on page 266.

Table 8-1 Mapping: Property/field to Particle for Element Wrapper

{min occurs}	if `@XmlElementWrapper.nillable()` is true or `@XmlElementWrapper.required()` is true, then 1; otherwise 0
{max occurs}	1
{term}	must be mapped as specified in Table 8-24, “Mapping: Property/field to Element Declaration for Element Wrapper,” on page 266.

Table 8-2 Mapping: Property/field to Element Declaration for Element Wrapper

{name}	if `@XmlElementWrapper.name()` is “`##default`“, then the XML name derived from the property or field name as specified in Section 8.12.1, “Java Identifier To XML Name”; otherwise `@XmlElementWrapper.name()`
{target namespace}	if `@XmlElementWrapper.namespace()` is “`##default` “, if the enclosing package has `@XmlSchema` annotation and is `@XmlSchema.elementFormDefault` is `@XmlNsForm.QUALIFIED`, then the namespace of the enclosing class. otherwise ““ (which produces unqualified element in the default namespace). otherwise `@XmlElementWrapper.namespace()`

Table 8-2 Mapping: Property/field to Element Declaration for Element Wrapper

{type definition}	if property/field is annotated with @XmlElementRef or @XmlElementRefs then the schema type as specified in Table 8-21, "Mapping: List of element instances to choice particle," on page 264 otherwise if property/field is annotated with @XmlElement or @XmlElements then the schema type as specified inTable 8-15, "Mapping: List of types to choice particle," on page 260.
{scope}	complex type to which the property's or the field's containing class is mapped.
{value constraint}	absent
{nillable}	@XmlElementWrapper.nillable()
{identity-constraint definitions}	absent
{substitution group affiliation}	absent
{substitution group exclusions}	{extension, restriction}
{disallowed substitution}	{extension, restriction,substitution}
{abstract}	false
{annotation}	unspecified

Table 8-3 Mapping: Property/field Element Wrapper with ref attribute

{min occurs}	1
{max occurs}	1
{term}	element declaration whose {name} is @XmlElementWrapper.name() and {target namespace} is @XmlElementWrapper.namespace(). ***Note*:** The element declaration is assumed to already exist and is not created.

8.9.6 @XmlAnyElement

8.9.6.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD})
public @interface XmlAnyElement {
    boolean lax() default false;
    Class<? extends DomHandler> value() default W3CDomHandler.class;
}
```

8.9.6.2 Mapping

The following mapping constraints must be enforced:

- The only other JAXB annotations allowed with `@XmlAnyElement` are: `@XmlElementRefs.`
- There must be only one property or field that is annotated with `@XmlAnyElement`.
- If a *baseType* has a property annotated with `@XmlAnyElement`, then no other sub type in the inheritance hierarchy rooted at *baseType* can contain a property annotated with `@XmlAnyElement`.

The property or field must be mapped as specified in Table 8-26, “Mapping: Wildcard schema component for wildcard (xs:any),” on page 268”.

Table 8-1 Mapping: Wildcard schema component for wildcard (xs:any)

`{namespace constraint}`	`##other`
`{process contents}`	`"lax"` if `lax()` is `true` otherwise `"skip"`
`{annotation}`	`unspecified`

8.9.7 @XmlAttribute

`@XmlAttribute` is used to map a property or a field to an XML attribute.

8.9.7.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD})
public @interface XmlAttribute {
```

```
    String name() default ##default";
    boolean required() default false;
    String namespace() default "##default" ;
}
```

8.9.7.2 Mapping

The following mapping constraints must be enforced:

- If the type of the field or the property is a collection type, then the collection item type must be mapped to schema simple type. Examples:

```
@XmlAttribute List<Integer> foo; // legal
@XmlAttribute List<Bar> foo; // illegal if Bar does not map to a
                             // schema simple type
```

- If the type of the field or the property is a non collection type, then the type of the property or field must map to a simple schema type. Examples:

```
@XmlAttribute int foo; // legal
@XmlAttribute Foo foo; // illegal if Foo does not map to a schema
                       // simple type
```

- The only additional mapping annotations allowed with `@XmlAttribute` are: `@XmlID`, `@XmlIDREF`, `@XmlList`, `@XmlSchemaType`, `@XmlValue`, `@XmlAttachmentRef`, `@XmlMimeType`, `@XmlInlineBinaryData`, `@XmlJavaTypeAdapter`.

Design Note – The mapping below supports mapping to either a local attribute or a reference to a global attribute that already exists. The latter is useful for mapping to attributes in foreign namespaces for e.g. <xs:attribute ref=”xml:lang”/>. Note that the attribtue is never created in the namespace, `@XmlAttribute.namespace()`; it is assumed to exist (for e.g. “xml:lang”)

The property or field is mapped to an attribtue reference when `@XmlAttribute.namespace()` is different from the `{target namespace}` of the type containing the property or field being mapped.

The property or field must be mapped as follows:

- If `@XmlAttribute.namespace()` is not “##default” and differs from the `{target namespace}` of the schema type to which the type containing the property or field is mapped, then the property or field

must be mapped as specified in Table 8-27, “Mapping: Property/field to Attribute Use (with ref attribute),” on page 270.

- otherwise, it must be mapped as specified in Table 8-28, “Mapping: Property/field to Attribute Use (no ref attribute),” on page 270.

Table 8-1 Mapping: Property/field to Attribute Use (with ref attribute)

`{required}`	`@XmlAttribute.required()`
`{attribute declaration}`	attribute declaration whose `{name}` is `@XmlAttribute.name()` and `{target namespace}` is `@XmlAttribute.namespace()`. For e.g. `<xs:attribute ref="xml:lang"/>`
`{value constraint}`	`absent`
`{annotation}`	`unspecified`

Table 8-2 Mapping: Property/field to Attribute Use (no ref attribute)

`{required}`	`@XmlAttribute.required()`
`{attribute declaration}`	Mapped as specified in Table 8-29, “Mapping: Property/field to Attribute Declaration,” on page 271”
`{value constraint}`	if field has access modifiers public and static then the `fixed` otherwise `absent`

Table 8-3 Mapping: Property/field to Attribute Declaration

{name}	if @XmlAttribute.name() is "##default", then the XML name derived from the property or field name as specified in Section 8.12.1, "Java Identifier To XML Name"; otherwise @XmlAttribute.name().
{target namespace}	if @XmlAttribute.namespace() is "##default", then value of targetNamespace in Table 8-3, "Mapping: Package to XML target namespace," on page 234; otherwise @XmlType.namespace()
{type definition}	if annotated with @XmlList, schema type derived by mapping as specified in Section 8.9.13, "@XmlList" otherwise if annotated with @XmlID, the schema type derived by mapping as specified in Section 8.9.11, "@XmlID" otherwise if annotated with @XmlIDREF, the schema type derived by mapping as specified in Section 8.9.12, "@XmlIDREF" otherwise if the type of the property is a collection type, then the schema type derived by mapping the collection item type. otherwise the schema type to which the type of the property is mapped.
{scope}	complex type of the containing class
{value constraint}	if field has access modifiers static and final then fixed otherwise absent
{annotation}	unspecified

8.9.8 XmlAnyAttribute

8.9.8.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD})
public @interface XmlAnyAttribute{}
```

8.9.8.2 Mapping

The following mapping constraints must be enforced:

- There must be only one property or field in a class that is annotated with `@XmlAnyAttribute.`
- The type of the property or the field must be `java.util.Map.`
- The only other annotations that can be used on the property or field with `@XmlAnyAttribute` are: `@XmlJavaTypeAdapter.`

The property or field must be mapped as specified in Table 8-30, “Mapping: Wildcard schema component for Attribute Wildcard,” on page 272.”

Table 8-1 Mapping: Wildcard schema component for Attribute Wildcard

{namespace constraint}	##other
{process contents}	skip
{annotation}	unspecified

8.9.9 @XmlTransient

`@XmlTransient` is used to prevent the mapping of a property or a field.

8.9.9.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD, TYPE})
public @interface XmlTransient {}
```

8.9.9.2 Mapping

The following mapping constraints must be enforced:

- The field or the property must not be mapped.
- `@XmlTransient` is mutually exclusive with all other mapping annotations.

8.9.10 @XmlValue

8.9.10.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD})
public @interface XmlValue {}
```

8.9.10.2 XmlValue Type Mapping

The following mapping constraints must be enforced:

1. At most one field or a property in a class can be annotated with `@XmlValue`.
2. `@XmlValue` can be used with the following annotations:
   - `@XmlList` - however this is redundant since `@XmlList` maps a type to a schema simple type that derives by list just as `@XmlValue` would.
   - `@XmlJavaTypeAdapter`
3. If the type of the field or property is a collection type, then the collection item type must map to a simple schema type. Examples:

```
// Examples (not exhaustive): Legal usage of @XmlValue
@XmlValue List<Integer> foo; // int maps to xs:int
@XmlValue String[] foo; // String maps to xs:string
@XmlValue List<Bar> foo; // only if Bar maps to a simple
                         // schema type
```

4. If the type of the field or property is not a collection type, then the type of the property or field must map to a schema simple type.
5. The containing class must not extend another class (other than java.lang.Obect).

8.9.10.3 Mapping

- If the type of the property or field is a collection type, then the type must be must be mapped as specified in Table 8-31, "@XmlValue: Mapping to list simple type," on page 274".
- Otherwise, the schema type to which the type of the property or field is mapped.

Table 8-1 @XmlValue: Mapping to list simple type

`{name}`	`absent`
`{target namespace}`	`{target namespace}` of the attribute or element to which the property or field is mapped and from where this type is referenced.
`{base type definition}`	ur-type definition, `xs:anyType`.
`{facets}`	`empty set`
`{fundamental facets}`	derived
`{final}`	#all
`{variety}`	list
`{item type definition}`	if the field, property or parameter is a collection type • if annotated with `@XmlIDREF`, then `xs:IDREF` as specified in Section 8.9.12, "@XmlIDREF" • otherwise the schema type to which the collection item type is mapped. otherwise • if annotated with `@XmlIDREF`, then `xs:IDREF` as specified in Section 8.9.12, "@XmlIDREF" • otherwise the schema type to which the type of the property, field or the parameter is mapped.
`{annotation}`	`unspecified`

8.9.11 @XmlID

8.9.11.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD})
public @interface XmlID {}
```

8.9.11.2 XmlID Type Mapping

The following mapping constraints must be enforced:

- at most one field or property in a class can be annotated with `@XmlID`.
- The type of the field or property must be `java.lang.String`.
- The only other program annotations allowed with `@XmlID` are: `@XmlAttribute` and `@XmlElement`.

The type of the annotated program element must be mapped to `xs:ID`.

8.9.12 @XmlIDREF

8.9.12.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD})
public @interface XmlIDREF {}
```

8.9.12.2 XmlIDREF Type Mapping

The following mapping constraints must be enforced:

- If the type of the field or property is a collection type, then the collection item type must contain a property or field annotated with `@XmlID`.
- If the field or property is not a collection type, then the type of the property or field must contain a property or field annotated with `@XmlID`.

  Note: If the collection item type or the type of the property (for non collection type) is `java.lang.Object`, then the instance must contain a property/field annotated with `@XmlID` attribute.

- The only additional mapping annotations allowed with `@XmlIDREF` are: `@XmlElement`, `@XmlAttribute, @XmlList,` and `@XmlElements, @XmlJavaTypeAdapter.`

If the type of the field or property is a collection type, then each collection item type must be mapped to `xs:IDREF`.

If the type of the field or property is single valued, then the type of the property or field must be mapped to `xs:IDREF`.

8.9.13 @XmlList

This annotation maps a collection type to a list simple type.

8.9.13.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD, PARAMETER})
public @interface XmlList {}
```

8.9.13.2 XmlIList Type Mapping

The following mapping constraints must be enforced:

- The type of the field, property or parameter must be a collection type.
- The collection item type must map to a simple schema type that does not derive by list. For example:

```
// Examples: Legal usage of @XmlList
@XmlList List<Integer> foo; // int maps to xs:int
@XmlList String[] foo; // String maps to xs:string
@XmlList List<Bar> foo; // only if Bar maps to a simple type

// Example: Illegal usage of @XmlList
public class Foo {
    // @XmlValue maps List to a XML Schema list simple type
    @XmlValue List<Integer> a;
}
class Bar {
    // Use of @XmlList is illegal since Foo itself mapped
    // to a XML Schema list simple type; XML Schema list simple
    // type can't derive from another XML Schema list simple type
    @XmlList List<Foo> y;
```

```
}
```

- The only additional mapping annotations allowed with `@XmlList` are: `@XmlElement, @XmlAttribute, @XmlValue` and `@XmlIDREF, @XmlJavaTypeAdapter.`

The type of the property or field must be mapped as specified in Table 8-32, "@XmlList: Mapping to list simple type," on page 277.

Table 8-1 @XmlList: Mapping to list simple type

{name}	absent
{target namespace}	{target namespace} of the attribute or element to which the property or field is mapped and from where this type is referenced.
{base type definition}	ur-type definition, xs:anyType.
{facets}	empty set
{fundamental facets}	derived
{final}	#all
{variety}	list
{item type definition}	if annotated with @XmlIDREF, then xs:IDREF as specified in Section 8.9.12, "@XmlIDREF" otherwise the schema type to which the collection item type is mapped.
{annotation}	unspecified

8.9.14 @XmlMixed

This annotation is used for dealing with mixed content in XML instances.

8.9.14.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD, METHOD})
public @interface XmlMixed {}
```

8.9.14.2 Mapping

The following mapping constraints must be enforced:

- The only additional mapping annotations allowed with `@XmlMixed` are: `@XmlElementRef, @XmlAnyElement, @XmlJavaTypeAdapter.`

The java.lang.String instances must be serialized as XML infoset text information items.

8.9.15 @XmlMimeType

8.9.15.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD,METHOD,PARAMETER})
public @interface XmlMimeType {
    // Textual representation of the MIME type, such as "image/jpeg"
    // "image/*", "text/xml; charset=iso-8859-1" and so on.
    String value();
}
```

8.9.15.2 Mapping

Table 8-1 @XmlMimeType: Mapping to Foreign Namespace attribute

{name}	“expectedContentTypes”
{target namespace}	"http://www.w3.org/2005/05/xmlmime"
attribute value	@XmlMimeType.value()

8.9.16 @XmlAttachmentRef

8.9.16.1 Synopsis

```
@Retention(RUNTIME) @Target({FIELD,METHOD,PARAMETER})
public @interface XmlAttachmentRef { }
```

8.9.16.2 Mapping

The type of property or field must map to `ref:swaRef`.

8.9.17 XmlInlineBinaryData

```
@Retention(RUNTIME) @Target({FIELD,METHOD,TYPE})
public @interface XmlInlineBinaryData {
}
```

8.9.17.1 Mapping

This annotation does not impact the schema generation. See the javadoc for `javax.xml.bind.annotation.XmlInlineBinaryData` for more details.

8.10 ObjectFactory Method

The annotations in this section are intended primarily for use by schema compiler in annotating element factory methods in the schema derived ObjectFactory class (Section 5.2, “Java Package”). They are not expected to be used when mapping existing classes to schema.

8.10.1 @XmlElementDecl

8.10.1.1 Synopsis

```
@Retention(RUNTIME) @Target({METHOD})
public @interface XmlElementDecl {
    Class scope() default GLOBAL.class;

    // XML namespace of element
    String namespace() default "##default";

    String name(); // local name of element
```

```
        //XML namespace name of a substitution group's head element.
        String substitutionHeadNamespace() default "##default";

        //XML local name of a substitution group's head element.
        String substitutionHeadName() default "";
        public final class GLOBAL {}
    }
```

8.10.1.2 Mapping

The following mapping constraints must be enforced:

- annotation can only be used on an *element factory method* (Section 5.2, “Java Package”). The annotation creates a mapping between an XML schema element declaration and a element factory method that returns a `JAXBElement` instance representing the element declaration. Typically, the element factory method is generated (and annotated) from a schema into the `ObjectFactory` class in a Java package that represents the binding of the element declaration's target namespace. Thus, while the annotation syntax allows `@XmlElementDecl` to be used on any method, semantically its use is restricted to annotation of element factory method
- class containing the element factory method annotated with `@XmlElementDecl` must be annotated with `@XmlRegistry`.
- element factory method must take one parameter assignable to `java.lang.Object`.
- two or more element factory methods annotated with `@XmlElementDecl` must not map to element declarations with identical `{name}` `{target namespace}` values.
- if type Foo has an element factory method and is also annotated with @XmlRootElement, then they must not map to element declarations with identical `{name}` and `{target namespace}` values.

  One example of where the above scenario occurs is when a developer attempts to add behavior/data to code generated from schema. For e.g. schema compiler generates an element instance factory method (e.g. `createFoo`) annotated with `@XmlElementDecl`. But the developer annotates `Foo` with `@XmlRootElement`.

An element factory method must be mapped as specified in Table 8-34, “Mapping: Element Factory method to Element Declaration,” on page 281.

Table 8-1 Mapping: Element Factory method to Element Declaration

{name}	@XmlElementDecl.name()
{target namespace}	if @XmlElementDecl.namespace() is "##default", then the value of the targetNamespace to which the package of the class containing the factory method is mapped as specified in Table 8-3, "Mapping: Package to XML target namespace," on page 234 otherwise @XmlElementDecl.namespace()
{type definition}	schema type to which the class is mapped as specified in Section 8.7.1, "@XmlType".
{scope}	global if @XmlElementDecl.scope() is @XmlElementDecl.GLOBAL otherwise the complex type definition to which the class containing the object factory method is mapped.
{value constraint}	absent
{nillable}	false
{identity-constraint definitions}	empty set
{substitution group affiliation}	element declaration derived from @XmlElementDecl.name() and @XmlElementDecl.substitutionHeadName()
{substitution group exclusions}	{}
{disallowed substitution}	{}
{abstract}	false
{annotation}	unspecified

8.11 Adapter

8.11.1 XmlAdapter

```
public abstract class XmlAdapter<ValueType,BoundType> {
    // Do-nothing constructor for the derived classes.
    protected XmlAdapter() {}

    // Convert a value type to a bound type.
    public abstract BoundType unmarshal(ValueType v);

    // Convert a bound type to a value type.
    public abstract ValueType marshal(BoundType v);
}
```

For an overview, see the section, Section 8.2.6, “Adapter”.

For detailed information, see the javadocs for `javax.xml.bind.annotation.adapters.XmlAdapter` and `javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter`.

8.11.2 @XmlJavaTypeAdapter

8.11.2.1 Synopsis

```
@Retention(RUNTIME)@Target({PACKAGE,FIELD,METHOD,TYPE,PARAMETER})
public @interface XmlJavaTypeAdapter {
    Class<? extends XmlAdapter> value();
    Class type() default DEFAULT.class;
    static final class DEFAULT {}
}
```

For an overview, see Section 8.2.6, “Adapter”.

8.11.2.2 Scope

The scope of `@XmlJavaTypeAdapter` must cover the program elements as specified below:

package:

For clarity, the following code example is used along with normative text.

```
// Adapts Foo type to MyFoo type
FooAdapter extends XmlAdapter<MyFoo, Foo>

// FooAdapter is installed at the package level - example.po
@XmlJavaTypeAdapter(value=FooAdapter.class, type=Foo.class)
```

A `@XmlJavaTypeAdapter` that extends `XmlAdapter <valueType, boundType>` and is specified as a package level annotation must adapt `boundType` at the point of reference as follows:

1. a property/field/parameter within a class in package (e.g `exmple.po`) whose reference type is `boundType`. For e.g.

   ```
   // Foo will be adapted to MyFoo
   Foo foo;
   ```

2. a property/field/parameter within a class in package (e.g `exmple.po`), where `boundType` is used as a parametric type. For e.g.

   ```
   // List<Foo> will be adapted to List<MyFoo>
   Foo foo;
   ```

class, interface, enum type:

For clarity, the following code example is used along with normative text.

```
// Adapts Foo type to MyFoo type
FooAdapter extends XmlAdapter<MyFoo, Foo>

// FooAdapter is specified on class, interface or enum type.
@XmlJavaTypeAdapter(FooAdapter.class)
public class Foo {...}
```

A `@XmlJavaTypeAdapter` that extends `XmlAdapter <valueType, boundType>` and is specified on the class, interface or Enum type (i.e. on a program element that matches meta annotation @Target={type}) must adapt `boundType` at the point of reference as follows:

1. a property/field whose reference type is `boundType`. For e.g.

   ```
   // Foo will be adapted to MyFoo
   ```

```
    Foo foo;
```

2. a property/field where `boundType` is used as a parametric type. For e.g.

```
    // List<Foo> will be adapted to List<MyFoo>
    List<Foo> foo;
```

Note: A `@XmlJavaTypeAdapter` on a class does not apply to references to it's sub class.

```
//Example:
@XmlJavaTypeAdapter(..) public class Foo {..}
...
public class DerivedFoo extends Foo {..}
...
public class Bar {
    // XmlJavaTypeAdapter applies to foo;
    public Foo foo;
    ...
    // XmlJavaTypeAdaper DOES NOT apply to derivedFoo;
    public DerivedFoo derivedFoo;
}
```

property/field/parameter:

A `@XmlJavaTypeAdapter` that extends `XmlAdapter <valueType, boundType>` and is specified on the property/field or parameter must adapt `boundType` as follows:

1. property/field is a single valued and its type is `boundType`:

```
    // Foo will be adapted to MyFoo
    @XmlJavaTypeAdapter(FooAdapter.class) Foo foo;
```

2. a property/field where `boundType` is used as a parametric type. For e.g.

```
    // List<Foo> will be adapted to List<MyFoo>
    List<Foo> foo;
```

8.11.2.3 Relationship to other annotations

`@XmlJavaTypeAdapter` must be applied first before any other mapping annotation is processed. Further annotation processing is subject to their respective mapping constraints. For example,

```
// PtoQAdapter is applied first and therefore converts type Q to P
```

```
// Next foo is mapped with a type of P (not Q) subject to the
// mapping constraints specified in @XmlElements.
@XmlJavaTypeAdapter(PtoQAdapter)
@XmlElements({
  @XmlElement(name="x",type=PX.class),
  @XmlElement(name="y",type=PY.class)
})
Q foo;

@XmlType abstract class P {}
@XmlType class PX extends P {}
@XmlType class PY extends P {}
```

8.11.2.4 Class Inheritance Semantics

When annotated on a class, the use of `@XmlJavaTypeAdapter` annotation is subject to the class inheritance semantics described here. The semantics is described in terms of two classes: a `BaseClass` and a `SubClass` that derives from `BaseClass`. There are two cases to consider:

- `@XmlJavaTypeAdapter` annotates the `BaseClass`
- `@XmlJavaTypeAdapter` annotates the `SubClass`, a class that derives from `BaseClass`.

BaseClass: In this case, `@XmlJavaTypeAdapter` annotates the `BaseClass`. In this case, the marshalling and unmarshalling of an instance of property or a field with a static type of baseClass must follow the schema to which `XmlJavaTypeAdapter.value()` is mapped.

```
//Example: code fragment
@XmlJavaTypeAdapter(..) BaseClass {..}
public SubClass extends BaseClass {..}
public BaseClass foo;
public SubClass subFoo = new SubClass();
foo = subFoo;
```

In the absence of `@XmlJavaTypeAdapter` annotation, the instance of subFoo is marshalled with an xsi:type:

```
<foo xsi:type="subClass"/>
```

With the `@XmlJavaTypeAdapter` annotation, however, the instance of subFoo must be marshalled/unmarshalled following the XML schema for `@XmlJavaTypeAdapter.value()`.

Subclass: In this case, `@XmlJavaTypeAdapter` annotates the `SubClass`. By definition, the annotation does not cover references to BaseClass. Thus, the schema types to which `SubClass` and `BaseClass` map are not in the same schema type hierarchy. Hence an object with a static type of `BaseClass` but containing an instance of `SubClass` can't be marshalled or unmarshalled. An attempt to do so must fail. For e.g,

```
// Example: Code fragment
BaseClass{..}
...
@XmlJavaTypeAdapter(..) SubClass extends BaseClass {..}

public class Bar {
    public BaseClass foo;
    public SubClass  subFoo = new SubClass();

    // marshal, unmarshal of foo will fail
    foo = subFoo;

    // marshal, unmarshal of subFoo will succeed
}
```

8.11.3 @XmlJavaTypeAdapters

This annotation is a container annotation for defining multiple `@XmlJavTypeAdapters` annotations at the package level.

8.11.3.1 Synopsis

```
@Retention(RUNTIME) @Target({PACKAGE})
public @interface XmlJavaTypeAdapters {
    // Collection of @{@link XmlJavaTypeAdapter} annotations
    XmlJavaTypeAdapter[] value();
}
```

8.11.3.2 Mapping

Each `@XmlJavaTypeAdapter` annotation in `@XmlJavaTypeAdapters.value()` must be mapped as specified in Section 8.11.2, "@XmlJavaTypeAdapter".

8.12 Default Mapping

This section describes the default mapping of program elements. The default mapping is specified in terms of *default annotations* that are considered to apply to a program element even in their absence.

8.12.1 Java Identifier To XML Name

The following is the default mapping for different identifiers:

- *class name*: a class name is mapped to an XML name by de capitalization using `java.beans.Introspector.decapitalize(`*`class name`*` )`.
- *enumtype name*: enumtype name is mapped to an XML name by de capitalization using `java.beans.Introspector.decapitalize(`*`enumtype name)`*.
- A property name (e.g. address) is derived from JavaBean access method (e.g. getAddress) by JavaBean de capitalization of the JavaBean property name `java.beans.Introspector.decapitalize(`*`JavaBeanAccessMethod`*`)`

8.12.2 Package

A package must be mapped with the following default package level mapping annotations:

- `@XmlAccessorType(`

```
javax.xml.bind.annotation.XmlAccessType.PUBLIC_M
EMBER)
```

- `@XmlAccessorOrder( javax.xml.bind.annotation.XmlAccessOrder.UNDEFIN ED)`

Design Note – Ordering of properties/fields based on source code order rather than alphabetical order is more useful. However, at this time there is no portable way to specify source code order. Order is undefined by Java reflection. Thus the default order has been chosen to be UNDEFINED. For applications which wish to remain portable across JAXB Providers, either XmlAccessOrder.ALPHABETICAL or @XmlType.propOrder() can be used.

- `@XmlSchema`

- properties and fields, unless explicitly annotated, must be considered to be annotated with `@XmlTransient`.

If the value of `@XmlAccessorType.value()` is `javax.xml.bind.annotation.XmlAccessType.PROPERTY`, then

- properties not explicitly annotated must be mapped; fields, unless explicitly annotated, must be considered to be annotated with `@XmlTransient.`

If the value of `@XmlAccessorType.value()` is `javax.xml.bind.annotation.XmlAccessType.FIELD`, then

- fields not explicitly annotated must be mapped; properties, unless explicitly annotated, must be considered to be annotated with `@XmlTransient.`

If the value of `@XmlAccessorType.value()` is `javax.xml.bind.annotation.XmlAccessType.PUBLIC_MEMBER`, then

- all properties and public fields, unless annotated with `@XmlTransient`, must be mapped.

See javadoc for `@javax.xml.bind.annotation.XmlAccessorType` for further information on inheritance rules for this annotation.

8.12.5.1 Default Mapping

A property name (e.g. address) must be derived from JavaBean access method (e.g. getAddress) by JavaBean decapitalization of the JavaBean property name `java.beans.Introspector.decapitalize(`*`JavaBeanAccessMethod`*`)`

A single valued property or field must be mapped with the following default mapping annotation:

```
@XmlElement
```

Note – An alternative to mapping property or a field to an element by default is to map property or field to an attribute if its type maps to a XML Schema simple type. However, neither alternative is dominant. The default has been chosen to be @XmlElement.

A property or field with a collection type must be mapped by with the following default mapping annotation:

- if the property or field is annotated with `@XmlList`, then the default mapping annotation is:
  `@XmlElement`

- otherwise the default mapping annotation is:
  `@XmlElements({ @XmlElement(nillable=true)})`

8.12.6 Map

By default, `java.util.Map<K,V>` must be mapped to the following anonymous schema type. The parameterized types K and V must be mapped as specified in Section 8.5.3.2, “Type Arguments and Wildcards”. The anonymous schema type is at the point of reference.

```
<!-- Default XML Schema mapping for Map<K,V> -->
<xs:complexType>
    <xs:sequence>
        <xs:element name="entry"
                    minOccurs ="0" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="key" type="xs:anyType"
                                           minOccurs="0"/>
                    <xs:element name="value" type="xs:anyType"
                                           minOccurs="0"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>

<!-- Default XML Schema mapping for Map<String, Integer>-->
<xs:complexType>
    <xs:sequence>
        <xs:element name="entry"
                    minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="key" type="xs:string"
                                           minOccurs="0"/>
```

```
                    <xs:element name="value" type="xs:int"
                                            minOccurs="0"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>

</xs:complexType>
```

The mapping of Map can be customized using `@XmlJavaTypeAdapter` annotation.

8.12.7 Multidimensional Array

By default, a multidimensional array must be mapped to a complex type as follows. Note the table specifies a two dimensional array mapping. If an array is more than two dimensions, then the mapping is used recursively.

Table 8-1 Mapping: Two dimensional array to Complex Type Definition

`{name}`	If the basetype is a primitive type (e.g. int[][]) or its corresponding wrapper class (e.g. Integer[][]), then the name is *basetype* concatenated with "Array" (e.g. intArray). otherwise if the basetype is a reference type (e.g. Foo[][]), then the XML name to which the reference type is mapped (e.g. foo) concatenated with "Array" (e.g. fooArray).
`{target namespace}`	if the *basetype* is a primitive or its corresponding wrapper class then `"http://jaxb.dev.java.net/array"` otherwise the namespace to which the reference type is mapped (e.g. for Foo[][], the namespace of the XML type to which Foo is mapped).
`{base type definition}`	`xs:anyType`
`{derivation method}`	`restriction`
`{final}`	`#all`

Table 8-1 Mapping: Two dimensional array to Complex Type Definition

{abstract}	false
{attribute uses}	empty set
{attribute wildcard}	absent
{content type}	element-only content content model mapped as specified in Table 8-36, “Mapping: Two dimensional array to sequence model group,” on page 292”.
{prohibited substitutions}	Empty set
{annotations}	absent

Table 8-2 Mapping: Two dimensional array to sequence model group

{compositor}	xs:sequence
{particles}	A repeating element defined as follows: <xs:element name="item" type=*schematype* minOccurs="0" maxOccurs="unbounded" nillable="true"/> where schematype is the schema to which the array’s component type is mapped (e.g. int[][], then “xs:int”; Foo[][] then “foo” assuming Foo is mapped to the schema type foo.
{annotation}	unspecified

8.13 Notes

This section contains a collection of notes intended to aid in the review of this version of the specification. They are collected here in a separate section and referenced from possibly multiple places elsewhere in the specification to make the specification more compact.

8.13.1 @XmlType: List simple type

It is possible to map a homogenous collection to a simple type with a variety of {list}. For e.g.

```
// Code fragment
public class USStateList {
    @XmlValue
    List <int> items;
}

// schema fragment
<xs:simpleType name="USStateList">
    <xs:list itemType="int"/>
<xs:simpleType>
```

Other types which can be mapped to a list simple type include: indexed property, single dimensional arrays.

CHAPTER 9

COMPATIBILITY

This section describes conformance requirements for an implementor of this specification. A JAXB implementation must implement these constraints, without exception, to provide a predictable environment for application development and deployment.

This section explicitly lists the high level requirements of this specification. Additional requirements can be found in other sections of this specification and the associated javadoc for package `javax.xml.bind` and its subpackages. If any requirements listed here conflict with requirements listed elsewhere in the specification, the requirements here take precedence and replace the conflicting requirements.

A JAXB implementation must implement the processing model specified in Appendix B, “Runtime Processing”.

A JAXB implementation included in a product that supports software development must support a schema generator. A schema generator must support all the Java Types to XML Schema mapping specified in Section 8, “Java Types To XML”.

A JAXB implementation included in a product that supports software development must support a schema compiler. All operating modes of a schema compiler must support all the XML Schema-to-Java bindings described in this specification. Additionally, any operating mode must not implement a default binding for XML Schema-to-Java bindings as an alternative to those specified in Section 6, “Binding XML Schema to Java Representations” nor alternative interpretations for the standard customizations described in Section 7, “Customizing XML Schema to Java Representation Binding.”

The default operating mode for a schema compiler MUST report an error when extension binding declaration is encountered. All operating modes for a schema compiler MUST report an error if an invalid binding customization is detected

as defined in Section 7. An extension binding declaration must be introduced in the following cases:

1. to alter a binding customization that is allowed to be associated with a schema element as specified in Section 7, “Customizing XML Schema to Java Representation Binding.”
2. **to associate a binding customization with a schema element where it is disallowed as specified in Section 7, “Customizing XML Schema to Java Representation Binding.”**

The default operating mode for a schema compiler MUST report an error when processing a schema that does not comply with the 2001 W3C Recommendation for XML Schema, [XSD Part 1] and [XSD Part 2].

A schema compiler MAY support non-default operating modes for binding schema languages other than XML Schema.

A schema compiler MUST be able to generate Java classes that are able to run on at least one Sun's Reference Implementation of the J2SE Java Runtime Environment that is Java SE 5 or higher.

A schema generator MAY support non-default operating modes for mapping Java types to schema languages other than XML Schema.

A Java platform configured with any JAXB 2.0 implementation MUST allow any JAXB 1.0 application, packaged as specified by the JAXB 1.0 implementation used by the application, to run as specified by the JAXB 1.0 specification. A JAXB 1.0 application uses schema-derived classes generated by a JAXB 1.0 compliant schema compiler and is distributed with the compatible JAXB 1.0 runtime implementation. Unlike JAXB 2.0, JAXB 1.0 specified a tight coupling between a JAXB 1.0 implementation’s schema-derived code and the JAXB 1.0 implementation of package `javax.xml.bind`. The required processing model for a JAXB 1.0 schema compiler generated package is specified in `javax.xml.bind.JAXBContext` class javadoc under the heading **SPEC REQUIREMENT**.

CHAPTER A

REFERENCES

[XSD Part 0] XML Schema Part 0: Primer,
Available at http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/
(schema fragments borrowed from this widely used source)

[XSD Part 1] XML Schema Part 1: Structures,
Available at http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[XSD Part 2] XML Schema Part 2: Datatypes,
Available at http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[XMl-Infoset] XML Information Set, John Cowan and Richard Tobin, eds., W3C, 16 March 2001. Available at http://www.w3.org/TR/2001/WD-xml-infoset-20010316/

[XML 1.0] Extensible Markup Language (XML) 1.0 (Second Edition),
W3C Recommendation 6 October 2000.
Available at http://www.w3.org/TR/2000/REC-xml-20001006.

[Namespaces in XML] Namespaces in XML
W3C Recommendation 14 January 1999.
Available at http://www.w3.org/TR/1999/REC-xml-names-19990114

[XPath], XML Path Language, James Clark and Steve DeRose, eds., W3C, 16 November 1999. Available at http://www.w3.org/TR/1999/REC-xpath-19991116

[XSLT 1.0] XSL Transformations (XSLT), Version 1.0, James Clark, W3C Recommendation 16 November 1999. Available at http://www.w3.org/TR/1999/REC-xslt-19991116.

[BEANS] JavaBeans(TM), Version 1.01, July 24, 1997. Available at `http://java.sun.com/beans`.

[XSD Primer] XML Schema Part 0: Primer,
W3C Recommendation 2 May 2001
Available at `http://www.w3.org/TR/xmlschema-0/`

[BLOCH] Joshua Bloch, Effective Java, Chapter 3, Typesafe Enums
`http://developer.java.sun.com/developer/Books/shiftintojavapage1.html#replaceenum`

[BLOCH_2] Joshua Bloch, Effective Java, Chapter 1, Item 1: Consider factory methods over constructors

[RFC2396] Uniform Resource Identifiers (URI): Generic Syntax, `http://www.ietf.org/rfc/rfc2396.txt`.

[JAX-RPC] Java[a] API for XML-based RPC JAX-RPC 1.0, `http://java.sun.com/xml/downloads/jaxrpc.html`.

[JAX-WS] Java[a] API for XML Web Services(JAX-WS) 2.0, Marc Hadley, Roberto Chinnici.

[JLS] or [JLS3] The Java Language Specification, 3rd Edition, Gosling, Joy, Steele, Bracha. Available at `http://java.sun.com/docs/books/jls`.

[NIST] NIST XML Schema Test Suite,
`http://xw2k.sdct.itl.nist.gov/xml/page4.html`.

[MTOM] SOAP Message Transmission Optimization Mechanism, http://www.w3.org/TR/2004/WD-soap12-mtom-20040608/

[XOP] Martin Gudgin, Noah Mendelsohn, Mark Nottingham, and Herve Ruellan. XML-binary Optimized Packaging. Recommendation, W3C, January 2005. http://www.w3.org/TR/xop10/.

[MIME] Anish Karmarkar, Ümit Yalçinalp, "Describing Media Content of Binary Data in XML", W3C note, http://www.w3.org/TR/2005/NOTE-xml-media-types-20050504

[WSIAP] Chris Ferris, Anish Karmarkar, and Canyang Kevin Liu. Attachments Profile Version 1.0. Final 1 Material, WS-I, August 2004. http://www.ws-i.org/Profiles/AttachmentsProfile-1.0-2004-08-24.html.

[WSIBP] WS-I Basic Profile 1.0, http://www.ws-i.org/Profile/Basic/2003-08/BasicProfile-1.0a.html

[CA] Common Annotations for the Java[a] Platform, Rajiv Mordani

APPENDIX **B**

RUNTIME PROCESSING

B.1 Introduction

Two of the important goals of JAXB 2.0 are portability (for inclusion in J2SE) and handling of invalid XML content (for e.g. schema evolution). These goals imply that a JAXB Provider must be capable of marshalling and unmarshalling XML instances using JAXB annotated classes derived using another JAXB Provider. To ensure portable behavior of JAXB mapped classes across JAXB Providers requires a specification of runtime processing model of the JAXB 2.0 binding framework.

This appendix specifies the runtime processing model of XML instances and requirements on JAXB 2.0 Provider's runtime. It differs from the documentation found elsewhere in the specification/javadocs. Chapter 4,"Binding Framework" and the javadocs for the package javax.xml.bind do describe the JAXB 2.0 binding framework. But these are written from a JAXB 2.0 developer perspective rather than from a JAXB 2.0 Provider perspective and thus do not describe requirements on JAXB 2.0 Provider runtime. This was sufficient for JAXB 1.0 since portability was not a goal for JAXB 1.0 and schema derived implementation classes were coupled to the JAXB 1.0 Provider runtime. However, this is insufficient for JAXB 2.0, where portability and handling invalid XML content are goals.

B.2 Scope and Conventions

B.2.1 Scope

This appendix describes marshalling and unmarshalling.

B.2.2 Format

The specification uses the following for specifying runtime behavior:

- XML Infoset, Second Edition, http://www.w3.org/TR/xml-infoset augmented with attributes xsi:type and xsi:nil that can be used by a document author.
- XML Schema abstract schema component model.
- JAXB defined annotations as needed

B.2.3 Notations

The following notations/conventions are used:

- For brevity, property is used but is used to mean JavaBean property or a field.
- XML Infoset properties that are associated with an information item are enclosed in [...], for e.g. AII.[local name]. And program elements the constitute the Java representation to which an information item are identified as for e.g. AII.valuetype.
- **AII: Attribute Information Item in XML Infoset**
- AII.[local name] : local name property in infoset for AII
- AII.[namespace] : namespace property in infoset for AII
- AII.[owner element] : owner element in infoset for AII
- AII.[normalized value]: normalized value in inforset for AII
- AII.property : JAXB property to which AII is mapped. The property in in the java type to which AII.[owner element] is mapped.
- AII.valuetype: Java type representing the XML serialized for AII.
- AII.boundtype: Java type that is bound; this differs from AII.valuetype only if JAXB property is associated with a @XmlJavaTypeAdapter.
- AII.schematype : schema type to which AII is bound statically using Java -> XML schema mapping rules.
- **EII: Element Information Item in infoset**
- EII.[local name] : local name property in XML infoset for EII

- EII.[namespace] : namespace property in XML infoset for EII
- EII.[children]: children property in XML infoset for EII
- EII.[parent]: parent property in XML infoset for EII
- EII.property : JAXB property to which EII is mapped. The property is in the javatype to which EII.[parent] is mapped.
- EII.valuetype : java type representing the XML serialized for EII
- EII.boundtype : java type that is bound; this differs from EII.valuetype only if JAXB property is associated with a @XmlJavaTypeAdapter.
- EII.schematype : schema type to which EII is bound statically using java -> XML schema mapping rules.
- EII.xsitype : the xsi:type specified in the XML instance by a document author. null if no xsi:type was specified.
- EII.xsinil : the xsi:nil specified in the XML instance by a document author. null if no xsi:nil was specified.

B.3 Unmarshalling

This section specifies the runtime behavior and JAXB provider requirements related to unmarshalling. The specification includes unmarshalling of invalid XML content in an XML instance.

This section specifies only flexible, standard unmarshalling (flexible because unmarshalling deals with invalid XML content). Other unmarshalling modes will not be specified. Flexible unmarshalling specified here must be supported by all JAXB Providers.

The unmarshalling methods in the binding framework fall into the following categories:

1. Unmarshal methods that do not take a declaredType as parameter:

   `javax.xml.bind.Unmarshaller.unmarshal`(...)
   `javax.xml.bind.Binder.unmarshal`(...)

2. **Unmarshal methods that take a declaredType as a parameter:**

```
javax.xml.bind.Unmarshaller.unmarshal(...,
java.lang.Class<T> declaredType)
javax.xml.bind.Binder.unmashal(...,
java.lang.Class<T> declaredType)
```

The unmarshal methods that do not take declaredType as parameter must be unmarshalled as specified in Section B.3.1, “Globally Declared Root Element”.

The unmarshal methods that take a `declaredType` as a parameter must be unmarshalled as specified in Section B.3.2, “Declared Type”.

B.3.1 Globally Declared Root Element

There are two ways that a root element can be represented in Java representation:

- as an element instance factory method that is generated in the public ObjectFactory class of a package when a schema is compiled. An element instance factory method is annotated with a

  @XmlElementDecl annotation. For e.g.

```
public class ObjectFactory {
    @XmlElementDecl(...)
    public JAXBElement<T> createFoo(T elementValue);
    ...
}
```

- as a type (either an enum type or a class) that has been annotated with @XmlRootElement. For e.g.

```
@XmlRootElement(...)
public class Foo {...}
```

The unmarshalling of XML content results in a content tree with a root that is an instance of either a `JAXBElement` instance or a type that is annotated with `@XmlRootElement`. The content tree must be created as follows:

1. lookup an element factory method in the ObjectFactory class matching on:

   EII.[namespace] == @XmlElementDecl.namespace() && EII.[local name] == @XmlElementDecl.name()

   or for a type annotated with @XmlRootElement matching on:

EII.[namespace] == @XmlRootElement.namespace() && EII.[local name] == @XmlRootElement.name()

NOTE: The lookup will only find one of the above not both. If both a type as well as an element factory method were found, it would be flagged as an error when JAXBContext is created.

2. **if an element factory method in the ObjectFactory class or a type annotated with @XmlRootElement is found, then determine the *valueType*.**

   a. if an element factory method is found, there is no @XmlJavaTypeAdapter associated with the value parameter to the element factory method, then the *valueType* is the java type of the value parameter to the element factory method. For e.g.

```
@XmlElementDecl(name = "bar", namespace = "")
    public JAXBElement<Foo> createBar(Foo value) {
    return new JAXBElement<Foo>(
        _Bar_QNAME, ((Class) Foo.class), null, value);
}
```

   the *valueType* type is Foo.

Note – *For ease of understanding the code generated by the Sun JAXB RI implementation has been shown above. But the implementation could be JAXB Provider dependent.*

---

   if the parameter is associated with @XmlJavaTypeAdapter, then the *valueType* is the java type specified in @XmlJavaTypeAdapter.value() .

   b. if a type annotated with @XmlRootElement is found then *valueType* is the type. For e.g.

```
@XmlRootElement(...)
public class Foo { ... }
```

Note – *@XmlRootElement and @XmlJavaTypeAdapter are mutually exclusive.*

   Go to step 4, “Check for type substitution”

3. **If neither the element factory method nor a type annotated with @XmlRootElement is found, then the element is unknown. Set *valueType* of the element to null.**

Even though the element is unknown, a document author can still perform type substitution. This case can arise if the XML schema contains only schema types and no global elements. For e.g a document author could have specified a xsi:type that has been mapped by JAXB. For e.g.

```
<unknownElement xsi:type="PurchaseOrder"/>
```

So goto step 4, "Check for type substitution"

4. **"Check for type substitution"**

   a. if xsi:type is not specified, and the *valueType* is null (i.e. the root element is unknown and we got to this step from step 3), throw a `javax.xml.bind.UnmarshalException` and terminate processing.

   b. otherwise, if `xsi:type` is specified, but is not mapped to a JAXB mapped type (e.g. class is not marked with @XmlType declaration), then throw a `javax.xml.bind.UnmarshalException` and terminate processing.

   c. otherwise, if xsi:type is specified, and is mapped to a JAXB mapped type set the *valueType* to the javatype to which xsi:type is mapped.

   d. otherwise, xsi:type is not specified; *valueType* is unchanged.

5. **Unmarshal *valueType* as specified in Section B.3.3, "Value Type".**

6. **If the element factory method is annotated with @XmlJavaTypeAdapter, then convert the *valueType* into a *boundType***

   ```
   boundType = @XmlJavaTypeAdapter.value().unmarshal(valueType)
   ```

7. **Determine the content root type to be returned by unmarshal() method.**

   a. if the element lookup returned an element instance factory method, then create a JAXBElement instance using the *boundType*. The content root type is the JAXBElement instance.

   b. otherwise, if the element lookup returned a type annotated with @XmlRootElement, then the content root type is the *boundType.*

   c. otherwise, the element is an unknown element. Wrap the *boundType* using JAXBElement with an element name in the XML instance document (e.g. "unknown Element"). The content root type is the JAXBElement instance.

8. **return the content root type.**

B.3.2 Declared Type

The unmarshalling process described in this section must be followed for the unmarshal methods that take a `declaredType` as a parameter.

1. Determine the *valueType* to be unmarshalled as follows:
   a. if `xsi:type` is specified, but is not mapped to a JAXB mapped type , then throw a `javax.xml.bind.UnmarshalException` and terminate processing.
   b. otherwise if `xsi:type` is specified and is mapped to JAXB mapped type, then *valueType* is the JAXB mapped type.
   c. otherwise *valueType* is the argument passed to `declaredType` parameter in the `unmarshal(..., java.lang.Class<T>declaredType )` call.
2. **Unmarshal *valueType* as specified in Section B.3.3, "Value Type".**

B.3.3 Value Type

The following steps unmarshal either EII.valuetype or AII.valuetype, depending upon whether an EII or AII is being unmarshalled.

Note: *Whether an EII or AII is being unmarshalled is determined by the "caller" of this section. AII.valuetype and EII.valuetype are assumed to be set by the time this section entered.*

1. If an instance of *valueType* does not exist, create an instance of *valueType* as follows (for e.g. if a value of a property with type `java.util.List` is non null, then unmarshal the value into that `java.util.List` instance rather than creating a new instance of `java.util.List` and assigning that to the property):
   a. if *valueType* is a class and is the type parameter specified in the element factory method, then instantiate the class using element factory method; otherwise instantiate the class using factory method if specified by `@XmlType.factoryClass()` and `@XmlType.factoryMethod();` or if there is no factory method, using the no-arg constructor.

b. if *valueType* is an enum type, then obtain an instance of the enum type for the enum constant annotated with `@XmlEnumValue` and `@XmlEnumValue.value()` matches the lexical representation of the EII.

2. Invoke any event callbacks in the following order as follows:

a. If *valueType* implements an unmarshal event callback `beforeUnmarshal(..)` as specified in Section 4.4.1,"Unmarshal Event Callback", then invoke `beforeUnmarshal(..)`.

b. If `Unmarshaller.getListener()` returns `Unmarshaller.Listener` that is not null, then invoke `Unmarshaller.Listener.beforeUnmarshal(..)`.

3. If an EII.valuetype is being unmarshalled, unmarshal into this instance the following. Note: The following steps can be done in any order; the steps are just broken down and listed separately for clarity:

If EII.valueType being unmarshalled

a. unmarshal each child element information item in EII.[children] as specified in Section B.3.4, "Element Information Item".

b. unmarshal each attribute information item in EII.[attributes] as specified in Section B.3.5, "Attribute Information Item".

4. Unmarshal the value of EII.schematype or AII.schematype following the Java to XML Schema rules defined in Chapter 8, "Java Types to XML". If the value in XML instance is unparseable, then it must be handled as specified in Section B.3.8.1, "Unparseable Data for Simple types".

5. Invoke any event callbacks in the following order as follows:

a. If *valueType* implements an unmarshal event callback `afterUnmarshal(..)` as specified in Section 4.4.1,"Unmarshal Event Callback", then invoke `afterUnmarshal(..)`.

b. If `Unmarshaller.getListener()` returns `Unmarshaller.Listener` that is not null, then invoke `Unmarshaller.Listener.afterUnmarshal(..)`.

6. return // either AII.valuetype or EII.valuetype.

B.3.4 Element Information Item

An EII must be unmarshalled as follows:

1. infer EII.property as specified in Section B.3.6.1, “Property Inference - Element Information Item”.
2. **if EII.property is null, then there is no property to hold the value of the element. If validation is on (i.e. Unmarshaller.getSchema() is not null), then report a javax.xml.bind.ValidationEvent. Otherwise, this will cause any unknown elements to be ignored.**

   If EII.property is not null and there is no setter method as specified in section Section B.5, “Getters/Setters” then report a javax.xml.bind.ValidationEvent.

   Goto step 8.
3. **infer the EII.valuetype as described in Section B.3.7.1, “Type Inference - Element Information Item”.**
4. **if EII.valuetype is null, then go to step 8.**

   NOTE: EII.valuetype = null implies that there was problem. so don't attempt to unmarshal the element.
5. **Unmarshal EII.valuetype as specified in Section B.3.3, “Value Type”.**
6. **if there is a @XmlJavaTypeAdapter associated with EII.property, then adapt the EII.valuetype as follows:**

   ```
   EII.boundtype =
       @XmlJavaTypeAdapter.value().unmarshal(EII.valuetype)
   ```

   otherwise

   EII.boundtype = EII.valuetype
7. **set the value of EII.property to EII.boundtype as follows:**

   Wrap EII.boundtype into a javax.xml.bind.JAXBElement instance if:

   a. the property is not a collection type and its type is javax.xml.bind.JAXBElement

   b. the property is a collection type and is a collection of JAXBElement instances (annotated with @XmlElementRef or @XmlElementRefs)

   If EII.property is not a collection type:

a. set the value of EII.property to EII.boundtype.

If EII.property is collection type:

a. add EII.boundtype to the end of the collection.

NOTE: Adding JAXBElement instance or a type to the end of the collection preserves document order. And document order could be different from the order in XML Scheme if the instance contains invalid XML content.

8. **return**

B.3.5 Attribute Information Item

An attribute information item must be unmarshalled as follows:

1. infer AII.property as described in section Section B.3.6.2, “Property Inference - Attribute Information Item”.
2. **if AII.property is null, then the attribute is invalid with respect to the XML schema. This is possible if for e.g. schema has evolved. If validation is on (i.e. Unmarshaller.getSchema() is not null), then report a javax.xml.bind.ValidationEvent. Otherwise, this will cause any unknown elements to be ignored.**

   If AII.property is not null and there is no setter method as specified in section Section B.5, “Getters/Setters” then report a javax.xml.bind.ValidationEvent.

   Goto step 8.
3. **infer the AII.valuetype as described in Section B.3.7.2, “Type Inference - Attribute Information Item”.**
4. **if AII.valuetype is null, then go to step 8.**

   NOTE: AII.valuetype = null implies that there was problem. so don't attempt to unmarshal the attribute.
5. **Unmarshal AII.valuetype as specified in Section B.3.3, “Value Type”.**
6. **If AII.property is associated with a `@XmlJavaTypeAdapter`, adapt AII.valuetype as follows:**

```
AII.boundtype =
       @XmlJavaTypeAdapter.value().unmarshal(AII.valuetype)
```

otherwise

AII.boundtype = AII.valuetype

7. **If AII.property is single valued:**

   a. set the value of AII.property to AII.boundtype.

   If AII.property is a collection type (e.g. List<Integer> was mapped to a Xml Schema list simple type using @XmlList annotation):

   add EII.boundtype to the end of the collection.

8. **return**

B.3.6 Property Inference

Unmarshalling requires the inference of a property or a field that contains the value of EII and AII being unmarshalled.

B.3.6.1 Property Inference - Element Information Item

The property to which an EII is mapped is inferred based on name.

Note – Inferring the property to which the EII is mapped by name rather than it's position in the content model within the schema is key to dealing with invalid XML content.

Infer EII.property by matching constraints described below:

1. initialize EII.property to null
2. **if property is mapped to XML Schema element declaration, elem, in the content model of EII.[parent].schematype && EII.[local name] == elem.{name} && EII.[namespace] == elem.{namespace}**

   set EII.property to property

   Goto step 4.

3. **If there is a JAXB property mapped to XML Schema wildcard (`xs:any`) (as determined by `@XmlAnyElement`), set this JAXB property to EII.property. This property will hold wildcard content (e.g. invalid XML content caused by schema evolution).**

4. **return EII.property**

B.3.6.2 Property Inference - Attribute Information Item

Infer the property for the AII by matching constraints described below:

1. initialize AII.property to null
2. **if property mapped to XML Schema attribute declaration, attr, in the content model of AII.[owner].schematype && AII.[local name] == attr.{name} && AII.[namespace] == attr.{namespace}**

   set AII.property to property

   Goto step 4.
3. **if there is a property mapped to a XML Schema xs:anyAttribute (i.e. annotated with `@XmlAnyAttribute`), then set this property to AII.property. This property holds XML content matching wildcard attribute (`xs:anyAttribute`) or unknown attributes (which can occur for e.g. if schema has evolved).**
4. **return AII.property**

B.3.7 Type Inference

Unmarshalling requires the inference of the type of a property or a field that to contain the value of EII and AII being unmarshalled.

B.3.7.1 Type Inference - Element Information Item

This section describes how to infer EII.valuetype; this holds the value of the element (content model + attributes).

EII.valuetype must be inferred as described below:

1. initialize EII.valuetype to null.
2. **if EII.xsitype is set, document author has performed type substitution.**

   Goto step 4 to handle type substitution.
3. **if EII.schematype is not mapped to a java type, then**

   a. report a validation event.

b. Go to step 7.

otherwise

a. set EII.valuetype to the javatype to which EII.schematype is mapped.

b. Go to step 7.

NOTE: This case can arise for example, when EII.schematype is compiled into a java type at schema compilation time, but the javatype was not registered with `JAXBContext.newInstance(..)`.

4. **check if EII.xsitype is mapped to a JAXB mapped type. It is possible that EII.xsitype is compiled to a javatype at schema compilation time, but the javatype was not registered with `JAXBContext.newInstance(..)`**

   If EII.xsitype is not mapped, then report a validation event.

   Goto step 7.

5. **check if the java type to which EII.xsitype is mapped is assignment comparable with the static type of the property/field if no `@XmlJavaTypeAdapter` is associated with the property/field or with the `valueType` specified in `XmlAdapter<valueType, boundType>` if a `@XmlJavaTypeAdapter` is associated with the property/field.**

   The above check can fail for e.g when a document author attempts to substitute a complex type that derives from simple type but customization to enable simple type substitution was not used. For e.g.

   a. <!-- local element with simple type -->

```
<xs:element name="foo" type="xs:int"/>
<!-- complex type for substituting the simple type -->
    <xs:complexType name="MyInt">
        <xs:extension xs:int>
            ...add attributes
        </xs:extends>
</xs:complexType>
```

   b. customization to handle type substitution of simple types is not used. So the property is

```
public int getFoo();
public void setFoo(int );
public class MyInt {...}
```

c. the document author attempts to substitute complexType MyInt.

```
<foo xsi:type="MyInt"/>
```

d. The type MyInt is not assignment comparable with int.

6. **set EII.valuetype to javatype to which EII.xsitype is mapped.**

   NOTE: if we got to this step, this implies that type substitution is valid.

7. **return EII.valuetype**

B.3.7.2 Type Inference - Attribute Information Item

Infer the AII.valuetype as follows:

1. initialize AII.valuetype to null.
2. **if AII.schematype is not mapped to a java type, then report a validation event. Otherwise, set AII.valuetype to the java type to which AII.schematype is mapped.**

   NOTE: This case can arise for example, when AII.schematype is compiled into a java type at schema compilation time, but the javatype is not registered with the `JAXBContext.newInstance(..)`
3. **return AII.valuetype**

B.3.8 Invalid XML Content

B.3.8.1 Unparseable Data for Simple types

If simple type data cannot be parsed into a java datatype, then the value of the java datatype must not change the current set value. An access to the datatype must return the value as specified in Section B.3.8.2, “Missing element information item”. If the conversion of lexical representation into a value results in an exception, then the exception must be caught and a validation event reported. This is to ensure that such conversion errors do not terminate unmarshalling.

```
// Example : XML Schema fragment
<xs:element name="foo" type="xs:int"/>

// Example: XML instance.
// Data is not parseable into type xs:int ; however unmarshal will
```

```
// still succeed.
<foo> SUN </foo>
```

B.3.8.2 Missing element information item

This case arises when an element declaration required by a XML schema is missing from the XML instance.

Property or field access must return the value specified in Section B.3.8.4, “Value for missing elements/attributes”

B.3.8.3 Missing Attribute

This case arises when a property or a field is mapped to an XML attribute but the attribute is missing from the XML instance.

Property or field access must return the value specified in Section B.3.8.4, “Value for missing elements/attributes”.

B.3.8.4 Value for missing elements/attributes

If an attribute or an element is missing from an XML instance, then unmarshal will not change the current set value. An access to the property will return the set value or if unset, the uninitialized value. The uninitialized value of the property or field depends upon it's type. If the type is

1. int - value is 0
2. **boolean - value is false**
3. **a reference (must be mapped to a simple type) - value is null.**
4. **float - the value is +0.0f**
5. **double - the value is 0.0d**
6. **short - the value is (short) 0**
7. **long - the value is 0L**

Note The uninitialized values are returned only if the value is not set. A value could be set for example in a validation handler that catches the validation event.

B.3.8.5 Unknown Element

In this case, XML instance contains EII for which there is no corresponding element declaration in the XML schema. If the valuetype to which the EII.parent maps contains a property/field annotated with `@XmlAnyElement`, this EII can be unmarshalled into the property/field.

Unknown attribute handling during unmarshalling is specified in Section B.3.6.1, “Property Inference - Element Information Item”.

B.3.8.6 Unknown attribute

In this case, XML instance contains AII for which there is no corresponding attribute declaration in the XML schema. If the valuetype to which the AII.parent maps contains a property/field annotated with `@XmlAnyAttribute`, the AII can be unmarshalled into the property/field.

Unknown attribute handling during unmarshalling is specified in Section B.3.6.2, “Property Inference - Attribute Information Item”.

B.4 Marshalling

To marshal a content tree, a JAXB application invokes one of the following marshal methods:

```
Marshaller.marshal(Object jaxbElement, ...) throws JAXBException;

Binder.marshal(Object jaxbObject, ...) throws JAXBException;
```

A JAXB Provider must marshal the content tree as follows:

- marshal the XML root element tag as specified in Section B.4.1, “XML Root Element Tag”
- marshal `obj` as specified in section Section B.4.2, “Type”.

B.4.1 XML Root Element Tag

1. If `obj` is an instance of `javax.xml.bind.JAXBElement` then marshal `obj` as specified in Section B.4.2.1, “JAXBElement”

Goto step 4

2. **If `obj.getClass()` is annotated with `@XmlRootElement`, then set {EII.[local name], EII.[namespace]} by deriving them from the @XmlRootElement annotation following the Java to Schema mapping rules in chapter 8. Marshal obj instance as specified in Section B.4.2, "Type".**

   Goto step 4

3. **If obj has neither an @XmlRootElement nor is a JAXBElement instance, then throw a `JAXBException` and terminate processing.**
4. **done**

B.4.2 Type

The type must be marshalled as follows. If the type is an instance of

- JAXBElement , then marshal as specified in Section B.4.2.1, "JAXBElement".
- Otherwise, marshal the type as follows. If the type is a :
  - class, then marshal as specified in Section B.4.2.2, "class".
  - primitive type or standard class, then marshal as specified in Section B.4.2.4, "Primitives and Standard classes"
  - enum type then marshal following the schema to which it is mapped.

B.4.2.1 JAXBElement

An `obj,` that is an instance of `javax.xml.bind.JAXBElement` must be marshalled as specified here:

1. `JAXBElement jaxbelem = (JAXBElement) obj;`
2. **set {EII.[local name] , EII.[namespace]} to `jaxbelem.getName()`**
3. **if `jaxbelem.isNil()` , add `xsi:nil` to EII.[attributes]**

Note – It is valid for a content model that is nil to have attributes. For e.g.

<foo xsi:nil attr1="1"/>

The attributes will be marshalled when the value that the JAXBElement wraps is marshalled.

4. **if `jaxbelem.isTypeSubstituted()` is true, then type substitution has occurred i.e. `jaxbelem.getDeclaredType()` (static type) is different from `jaxbelem.getValue()` (the type of the value for this instance). So,**

   a. EII.[local name] = "type"

   b. EII.[prefix]="xsi"

   c. EII.[normalized value] = QName of the schema type to which `jaxbelem.getValue()` is mapped following Java -> Schema mapping rules in Chapter 8. For e.g.

   ```
   <foo xsi:type="MyAddrType"/>
   ```

5. **set *boundType* to`jaxbelem.getValue()` if `jaxbelem.isTypeSubstituted()` is `true` otherwise `jaxbelem.getDeclaredType()`**

6. **determine the *valueType* to be marshalled. If the program element being processed is associated with @XmlJavaTypeAdapter then *boundType* is**

   ```
   valueType = @XmlJavaTypeAdapter.value().marshal(boundType)
   ```

   otherwise *valueType* is *boundType*

7. **map *valueType* to XML infoset information items as specified in Section B.4.2, "Type" and add them to EII.**

8. **marshal EII.**

B.4.2.2 class

A class must be mapped to XML infoset items as follows:

1. If a class mapped to a value as specified Section 8.9.10, "@XmlValue", then map the value to an XML infoset and add it to EII.[children]

   return

2. **For each property that is mapped to XML attribute as specified in Section 8.9.7, "@XmlAttribute":**

a. derive {AII.[local name], AII.[prefix], AII.[namespace] } from {name} {target namespace}.

b. AII.[normalized value] = value of property as specified in Section B.4.2.3, “property type”

c. add AII to EII.[attributes]

NOTE: There order in which the properties are marshalled is not specified (XML attributes are unordered by XML Schema).

3. For each property that is mapped to an XML element declaration, elem:

a. derive {childEII.[local name], childEII.[prefix], childEII.[namespace] } from elem.{name} elem.{target namespace}

b. map property type to XML infoset items into childEII as specified in Section B.4.2.3, “property type”.

c. add childEII to EII.[children]

B.4.2.3 property type

The value of a property with type , *boundType*, must be marshalled into childEII (set by “caller of this section”) as follows:

1. If property does not have a getter method as specified in section Section B.5, “Getters/Setters” then report a javax.xml.bind.ValidationEvent. Goto step 4.

2. If the value of the property being marshalled is a subtype *boundType,* then

a. EII.[local name] = “type”

b. EII.[prefix]=”xsi”

c. EII.[normalized value] = QName of the schema type to which `jaxbelem.getValue()` is mapped following Java -> Schema mapping rules in Chapter 8. For e.g.

```
<foo xsi:type=”MyAddrType”/>
```

d. add EII to childEII

3. Marshal the value as specified in Section B.4.2, “Type”.

4. Return

B.4.2.4 Primitives and Standard classes

Primitive values and standard classes described in this section map to XML schema simple types.

The value of a primitive type or a standard class must be marshalled to a lexical representation or unmarshalled from a lexical representation as specified in the below:

- using a print or parse method in javax.xml.bind.DatatypeConverter interface:

  Many of the types have a corresponding print and parse method in javax.xml.bind.DatatypeConverter interface for converting a value to a lexical representation in XML and vice versa. The implementation of DatatypeConverter is JAXB Provider specific.

  A XML Schema simple type can have more than lexical representation (e.g. “true” “false” “0” “1”). Since the DatatypeConverter implementation is JAXB Provider specific, the exact lexical representation that a value is marshalled to can vary from one JAXB Provider to another. However, the lexical representation must be valid with respect to the XML Schema.
- some data types such as XMLGregorianCalendar contain methods on the class that return or consume their XML lexical representation. For such datatypes, the method indicated in the table is used.
- A wrapper class (e.g. java.lang.Integer) must be converted to its non wrapper counterpart (e.g. int) and then marshalled.

Table 2-1 Lexical Representation of Standard Classes

Java Standard Classes	printMethod	parse Method
`java.lang.String`	`printString`	`parseString`
`java.util.Calendar`	`printDateTime`	`parseDateTime`
`java.util.Date`	`printDateTime`	`parseDateTime`
`java.net.URI`	`URI.toString()`	`URI(String str)`

Table 2-1 Lexical Representation of Standard Classes

Java Standard Classes	printMethod	parse Method
javax.xml.datatype.XMLGregorianCalendar	XMLGregorianCalendar.toXMLFormat()	DatatypeFactory.newXMLGregorianCalendar(String lexicalRepresentation)
javax.xml.datatype.Duration	Duration.toString()	DatatypeFactory.newDuration(String lexicalRepresentation)
java.util.UUID	UUID.toString()	UUID.fromString()

B.4.2.5 Null Value

A null value in Java representation can be marshalled either as an absence of an element from an XML instance or as `xsi:nil`. The marshalled value depends upon the values of `@XmlElement.required()` and `@XmlElement.nillable()` annotation elements on the property/field and must be marshalled as shown below. For clarity, example schema fragments (as determined by the mapping rules specified in Chapter 8) for the following field

```
@XmlElement(required=”..”, nillable=”...”)
foo;
```

are reproduced here along with the XML representation for null value produced by marshalling.

- `@XmlElement(required=true, nillable = false)`

  The value of the property/field cannot be null.

  ```
  // Example: generated schema
  <xs:element name=”foo” minOccurs=”1”/ ...>
     ...
  </xs:element>
  ```

- `@XmlElement(required=true, nillable=true)`

  null is marshalled as `xsi:nil=”true”`

```
// Example:generated schema
<xs:element name="foo" minOccurs="1" nillable="true" ...>
    ...
/xs:element>

<!-- marshalled XML representation for null value -->
<foo xsi:nil="true" .../>
```

- `@XmlElement(required=false, nillable=true)`

  null is marshalled as `xsi:nil="true"`

```
// Examle: generated schema
<xs:element name="foo" minOccurs="0" ...>
    ...
</xs:element>

<!-- Example: marshalled XML representation for null value -->
<foo xsi:nil="true" .../>
```

- `@XmlElement(required=false, nillable=false)`

  null is not marshalled i.e it maps to absence of an element from XML instance.

```
// Example: Generated schema
<xs:element name="foo" minOccurs="0" ...>
    ...
</xs:element>

<!-- Example: null value for foo not marshalled -->
```

B.5 Getters/Setters

When `@XmlAccessType.PUBLIC_MEMBER` or `@XmlAccessType.PROPERTY` is in effect for a class, then the instance of the class is marshalled using getter/setter methods as opposed to fields. This section outlines the constraints the must be checked at runtime. A constraint failure is handled as specified elsewhere in the chapter from where this section is referenced.

Unmarshalling : A property must have a setter method if

- `@XmlAccessorType.PUBLIC_MEMBER` or `@XmlAccessorType.PROPERTY` applies to the property.
- or if the property's getter/setter method is annotated with a mapping annotation.

The one exception to the above constraint is: if property type is `java.util.List` then only a getter method is required.

Design Note – For a JavaBean property with getter/setter methods, a setter method is required for unmarshalling when public API (as opposed to fields) is used i.e. either `@XmlAccessorType.PUBLIC_MEMBER` or `@XmlAccessorType.PROPERTY` is in effect. If `@XmlAccessorType.FIELD` is in effect, then unmarshalling is based on fields and hence a setter method is not required. There is however one exception.

When starting from schema, a schema component (e.g. a repeating occurrence of an element) can be bound to a `java.util.List` property (so modifications to `java.util.List` can be intercepted, a design decision from JAXB 1.0). Thus only in this case a setter method is not required. E.g.

```
public java.util.List getFoo();
// public void setFoo(..) not required
```

Marshalling: A property must have a getter method if

- `@XmlAccessType.PUBLIC_MEMBER` or `@XmlAccessType.PROPERTY` applies to the class
- or if the property's getter/setter method is annotated with a mapping annotation.

APPENDIX C

NORMATIVE BINDING SCHEMA SYNTAX

C.1 JAXB Binding Schema

Online versions of JAXB Binding Schema are available at:

```
http://java.sun.com/xml/ns/jaxb/bindingschema_2_0.xsd
http://java.sun.com/xml/ns/jaxb/bindingschema_1_0.xsd
```

Additions to JAXB 1.0 binding schema are in **bold**.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema
    targetNamespace = "http://java.sun.com/xml/ns/jaxb"
    xmlns:jaxb = "http://java.sun.com/xml/ns/jaxb"
    xmlns:xs = "http://www.w3.org/2001/XMLSchema"
    elementFormDefault = "qualified"
    attributeFormDefault = "unqualified">
<xs:group name = "declaration">
    <xs:choice>
      <xs:element ref = "jaxb:globalBindings"/>
      <xs:element ref = "jaxb:schemaBindings"/>
      <xs:element ref = "jaxb:class"/>
      <xs:element ref = "jaxb:property"/>
      <xs:element ref = "jaxb:typesafeEnumClass"/>
      <xs:element ref = "jaxb:typesafeEnumMember"/>
      <xs:element ref = "jaxb:javaType"/>
      <xs:element ref = "jaxb:dom"/>
      <xs:element ref = "jaxb:inlineBinaryData"/>
      <xs:any namespace = "##other" processContents = "lax"/>
    </xs:choice>
  </xs:group>
  <xs:attribute name = "version" type="xs:token" >
    <xs:annotation><xs:documentation>
```

```
Used to specify the version of the binding schema on the schema element for inline
annotations or jaxb:bindings for external binding.
    </xs:documentation></xs:annotation>
  </xs:attribute>
  <xs:attributeGroup name = "propertyAttributes">
    <xs:annotation>
      <xs:documentation>
Attributes used for property customization. The attribute group can be referenced
either from the globalBindings declaration or from the
property declaration. The following defaults are defined by the JAXB specification
in global scope only. Thus they apply when the propertyAttributes group is
referenced from the globalBindings declaration but not when referenced from the
property declaration.
            collectionType   a class that implements java.util.List.
            fixedAttributeAsConstantProperty  false
            enableFailFastCheck        false
            generateIsSetMethod        false
            optionalProperty           wrapper
            generateElementProperty    false
        </xs:documentation>
    </xs:annotation>
    <xs:attribute name = "collectionType" type="jaxb:referenceCollectionType"/>
    <xs:attribute name = "fixedAttributeAsConstantProperty" type = "xs:boolean"/>
    <xs:attribute name = "enableFailFastCheck" type = "xs:boolean"/>
    <xs:attribute name = "generateIsSetMethod" type = "xs:boolean"/>
    <xs:attribute name = "optionalProperty">
    <xs:simpleType>
        <xs:restriction base="xs:NCName">
            <xs:enumeration value="wrapper"/>
            <xs:enumeration value="primitive"/>
            <xs:enumeration value="isSet"/>
        </xs:restriction>
    </xs:simpleType>
    </xs:attribute>
    <xs:attribute name = "generateElementProperty" type="xs:boolean"/>
</xs:attributeGroup>
  <xs:attributeGroup name = "XMLNameToJavaIdMappingDefaults">
    <xs:annotation>
      <xs:documentation>Customize XMLNames to Java id mapping</xs:documentation>
    </xs:annotation>
    <xs:attribute name = "underscoreBinding" default = "asWordSeparator" type =
"jaxb:underscoreBindingType"/>
  </xs:attributeGroup>
  <xs:attributeGroup name = "typesafeEnumClassDefaults">
    <xs:attribute name = "typesafeEnumMemberName" default = "skipGeneration" type
= "jaxb:typesafeEnumMemberNameType"/>
    <xs:attribute name = "typesafeEnumBase" default = "xs:string" type =
"jaxb:typesafeEnumBaseType"/>
    <xs:attribute name = "typesafeEnumMaxMembers" type="xs:int" default="256"/>
  </xs:attributeGroup>
  <xs:element name = "globalBindings">
    <xs:annotation>
      <xs:documentation>Customization values defined in global scope.</
xs:documentation>
    </xs:annotation>
```

```
    <xs:complexType>
      <xs:sequence minOccurs = "0">
        <xs:element ref = "jaxb:javaType" minOccurs = "0" maxOccurs = "unbounded"/>
        <xs:element ref = "jaxb:serializable" minOccurs = "0"/>
        <xs:any namespace = "##other" processContents = "lax">
          <xs:annotation> <xs:documentation>
            allows extension binding declarations to be specified.
          </xs:documentation></xs:annotation>
        </xs:any>
      </xs:sequence>
      <xs:attributeGroup ref = "jaxb:XMLNameToJavaIdMappingDefaults"/>
      <xs:attributeGroup ref = "jaxb:typesafeEnumClassDefaults"/>
      <xs:attributeGroup ref = "jaxb:propertyAttributes"/>
      <xs:attribute name="generateValueClass" type="xs:boolean"
            default= "true"/>
      <xs:attribute name="generateElementClass" type="xs:boolean"
            default= "false"/>
      <xs:attribute name="mapSimpleTypeDef" type="xs:boolean"
            default= "false"/>
    <xs:attribute name="localScoping" default= "nested">
    <xs:simpleType>
       <xs:restriction base="xs:NCName">
        <xs:enumeration value="nested"/>
        <xs:enumeration value="toplevel"/>
          </xs:restriction>
       </xs:simpleType>
      </xs:attribute>
        <xs:attribute name = "enableJavaNamingConventions" default = "true" type
= "xs:boolean"/>
        <xs:attribute name = "choiceContentProperty" default = "false" type =
"xs:boolean"/>
    </xs:complexType>
  </xs:element>
  <xs:element name = "schemaBindings">
    <xs:annotation>
      <xs:documentation>Customization values with schema scope<xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:all>
        <xs:element name = "package" type = "jaxb:packageType" minOccurs = "0"/>
        <xs:element name = "nameXmlTransform" type = "jaxb:nameXmlTransformType"
minOccurs = "0"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name = "class">
    <xs:annotation>
      <xs:documentation>Customize interface and implementation class.</
xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "javadoc" type = "xs:string" minOccurs = "0"/>
      </xs:sequence>
      <xs:attribute name = "name" type = "jaxb:javaIdentifierType">
```

```
        <xs:annotation>
          <xs:documentation>Java class name without package prefix.</
xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name = "implClass" type = "jaxb:javaIdentifierType">
        <xs:annotation>
          <xs:documentation>Implementation class name including package prefix.
</xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="generateValueClass" type="xs:boolean" default= "true"/>
    </xs:complexType>
  </xs:element>
  <xs:element name = "property">
    <xs:annotation>
      <xs:documentation>Customize property.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:all>
        <xs:element name = "javadoc" type = "xs:string" minOccurs="0"/>
       <xs:element name = "baseType" type="jaxb:propertyBaseType" minOccurs="0"/>
      </xs:all>
      <xs:attribute name = "name" type = "jaxb:javaIdentifierType"/>
     <xs:attribute name = "attachmentRef" default="default">
        <xs:simpleType>
            <xs:restriction base="xs:NCName">
                <xs:enumeration value="resolve"/>
                <xs:enumeration value="doNotResolve"/>
                <xs:enumeration value="default"/>
            </xs:restriction>
        </xs:simpleType>
     </xs:attribute>
      <xs:attributeGroup ref = "jaxb:propertyAttributes"/>
    </xs:complexType>
  </xs:element>
  <xs:element name = "javaType">
    <xs:annotation>
      <xs:documentation>Data type conversions;  overriding builtins</
xs:documentation>
    </xs:annotation>
    <xs:complexType>
     <xs:attribute name = "name" use = "required" type = "jaxb:javaIdentifierType">
        <xs:annotation>
          <xs:documentation>name of the java type to which xml type is to be
bound.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name = "xmlType" type = "xs:QName">
        <xs:annotation>
          <xs:documentation> xml type to which java datatype has to be bound.
Must be present when javaType is scoped to globalBindings.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
```

```
      <xs:attribute name = "parseMethod" type = "jaxb:javaIdentifierType"/>
      <xs:attribute name = "printMethod" type = "jaxb:javaIdentifierType"/>
        <xs:annotation>
          <xs:documentation>
If true, the parsMethod and printMethod must reference a method
signtature that has a second parameter of type NamespaceContext.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
<xs:element name = "typesafeEnumClass">
    <xs:annotation>
     <xs:documentation> Bind to a type safe enumeration class.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "javadoc" type = "xs:string" minOccurs = "0"/>
        <xs:element ref = "jaxb:typesafeEnumMember" minOccurs = "0" maxOccurs =
"unbounded"/>
      </xs:sequence>
      <xs:attribute name = "name" type = "jaxb:javaIdentifierType"/>
      <xs:attribute name = "map" type = "xs:boolean" default = "true"/>
    </xs:complexType>
  </xs:element>
  <xs:element name = "typesafeEnumMember">
    <xs:annotation>
      <xs:documentation> Enumeration member name in a type safe enumeration
class.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "javadoc" type = "xs:string" minOccurs = "0"/>
      </xs:sequence>
      <xs:attribute name = "value" type="xs:anySimpleType"/>
     <xs:attribute name = "name" use = "required" type = "jaxb:javaIdentifierType"/>
    </xs:complexType>
  </xs:element>

  <!-- TYPE DEFINITIONS -->

  <xs:complexType name = "propertyBaseType">
    <xs:all>
      <xs:element ref = "jaxb:javaType" minOccurs = "0"/>
    </xs:all>
      <xs:attribute name = "name" type = "jaxb:javaIdentifierType">
        <xs:annotation><xs:documentation>
```

```
The name attribute for [baseType] enables more precise control over the actual
base type for a JAXB property. This customization enables specifying a more
general/specialized base type than the property's default base type. The name
attribute value must be a fully qualified Java class name. Additionally, this Java
class must be a super interface/class or subclass of the default Java base type
for the property. When the default base type is a primitive type, consider the
default Java base type to be the Java wrapper class of that primitive type.This
customization is useful to enable simple type substitution for a JAXB property
representing with too restrictive of a default base type. It also can be used to
provide stronger typing for the binding of an element/attribute of type xs:IDREF.
    </xs:documentation></xs:annotation>
      </xs:attribute>
  </xs:complexType>
<xs:complexType name = "packageType">
    <xs:sequence>
      <xs:element name = "javadoc" type = "xs:string" minOccurs = "0"/>
    </xs:sequence>
    <xs:attribute name = "name" type = "jaxb:javaIdentifierType"/>
  </xs:complexType>
  <xs:simpleType name = "underscoreBindingType">
    <xs:annotation>
        <xs:documentation>
Treate underscore in XML Name to Java identifier mapping.
        </xs:documentation>
    </xs:annotation>
    <xs:restriction base = "xs:string">
      <xs:enumeration value = "asWordSeparator"/>
      <xs:enumeration value = "asCharInWord"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name = "typesafeEnumBaseType">
    <xs:annotation>
      <xs:documentation>
XML types or types derived from them which have enumeration facet(s) which are be
mapped to typesafeEnumClass by default. The following types cannot be specified
in this list: "xsd:QName", "xsd:base64Binary", "xsd:hexBinary", "xsd:date",
"xsd:time", "xsd:dateTime", "xsd:duration","xsd:gDay", "xsd:gMonth", "xsd:Year",
"xsd:gMonthDay", "xsd:gYearMonth", "xsd:ID", "xsd:IDREF", "xsd:NOTATION"
      </xs:documentation>
    </xs:annotation>
    <xs:list itemType = "xs:QName"/>
  </xs:simpleType>
  <xs:simpleType name = "typesafeEnumMemberNameType">
    <xs:annotation><xs:documentation>
Used to customize how to handle name collisions.
    </xs:documentation></xs:annotation>
    <xs:restriction base = "xs:string">
      <xs:enumeration value = "generateName"/>
      <xs:enumeration value = "generateError"/>
      <xs:enumeration value = "skipGeneration"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name = "javaIdentifierType">
    <xs:annotation>
```

```
      <xs:documentation>Placeholder type to indicate Legal Java identifier.</
xs:documentation>
    </xs:annotation>
    <xs:list itemType = "xs:NCName"/>
  </xs:simpleType>
  <xs:complexType name = "nameXmlTransformRule">
    <xs:annotation>
      <xs:documentation>Rule to transform an Xml name into another Xml name</
xs:documentation>
    </xs:annotation>
    <xs:attribute name = "prefix" type = "xs:string">
      <xs:annotation>
        <xs:documentation>prepend the string to QName.</xs:documentation>
      </xs:annotation>
    </xs:attribute>
    <xs:attribute name = "suffix" type = "xs:string">
      <xs:annotation>
        <xs:documentation>Append the string to QName.</xs:documentation>
      </xs:annotation>
    </xs:attribute>
  </xs:complexType>
  <xs:complexType name = "nameXmlTransformType">
    <xs:annotation><xs:documentation>
       Allows transforming an xml name into another xml name. Use case UDDI 2.0
schema.</xs:documentation></xs:annotation>
    <xs:all>
     <xs:element name = "typeName" type = "jaxb:nameXmlTransformRule" minOccurs =
"0">
       <xs:annotation><xs:documentation>Mapping rule for type definitions.</
xs:documentation></xs:annotation>
      </xs:element>
     <xs:element name = "elementName" type = "jaxb:nameXmlTransformRule" minOccurs
= "0">
       <xs:annotation><xs:documentation>Mapping rule for elements</
xs:documentation></xs:annotation>
      </xs:element>
      <xs:element name = "modelGroupName" type = "jaxb:nameXmlTransformRule"
minOccurs = "0">
       <xs:annotation>
         <xs:documentation>Mapping rule  for model group</xs:documentation>
       </xs:annotation>
      </xs:element>
      <xs:element name = "anonymousTypeName" type = "jaxb:nameXmlTransformRule"
minOccurs = "0">
       <xs:annotation>
       <xs:documentation>Mapping rule for class names generated for an anonymous
type.</xs:documentation>
       </xs:annotation>
      </xs:element>
    </xs:all>
  </xs:complexType>
  <xs:attribute name = "extensionBindingPrefixes">
    <xs:annotation>
      <xs:documentation>
A schema compiler only processes this attribute when it occurs on an
```

```
an instance of xs:schema element.  The value of this attribute is a
whitespace-separated list of namespace prefixes.  The namespace bound
to each of the prefixes is designated as a customization declaration
namespace.
      </xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:list itemType = "xs:normalizedString"/>
    </xs:simpleType>
  </xs:attribute>
  <xs:element name = "bindings">
    <xs:annotation>
      <xs:documentation>
Binding declaration(s) for a remote schema. If attribute node is set, the binding
declaraions are associated with part of the remote schema designated by
schemaLocation attribute. The node attribute identifies the node in the remote
schema to associate the binding declaration(s) with.
      </xs:documentation>
    </xs:annotation>
   <!-- a <bindings> element can contain arbitrary number of binding declarations
or nested <bindings> elements -->
    <xs:complexType>
      <xs:sequence>
        <xs:choice minOccurs = "0" maxOccurs = "unbounded">
          <xs:group ref = "jaxb:declaration"/>
          <xs:element ref = "jaxb:bindings"/>
        </xs:choice>
      </xs:sequence>
      <xs:attribute name = "schemaLocation" type = "xs:anyURI">
        <xs:annotation>
          <xs:documentation>
Location of the remote schema to associate binding declarations with.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name = "node" type = "xs:string">
        <xs:annotation>
          <xs:documentation>
The value of the string is an XPATH 1.0 compliant string that resolves to a node
in a remote schema to associate binding declarations with. The remote schema is
specified by the schemaLocation attribute occuring in the current element or in a
parent of this element. </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name = "version" type = "xs:token">
        <xs:annotation>
          <xs:documentation>
Used to indicate the version of binding declarations. Only valid on root level
bindings element. Either this or "jaxb:version" attribute but not both may be
specified. </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute ref = "jaxb:version">
        <xs:annotation>
          <xs:documentation>
```

```
Used to indicate the version of binding declarations. Only valid on root level
bindings element. Either this attribute or "version" attribute but not both may
be specified.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="referenceCollectionType">
    <xs:union>
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="indexed"/>
        </xs:restriction>
      </xs:simpleType>
      <xs:simpleType>
        <xs:restriction base="jaxb:javaIdentifierType"/>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
  <xs:element name="dom">
    <xs:complexType>
    <xs:attribute name = "type" type="xs:NCName" default="w3c">
        <xs:annotation><xs:documentation>Specify DOM API to bind to JAXB property
to.<xs:documentation></xs:annotation>
        </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="inlineBinaryData">
    <xs:annotation><xs:documentation>
Disable MTOM/XOP encoding for this binary data. Annotation can be placed on a type
defintion that derives from a W3C XSD binary data type or on an element that has a
type that is or derives from a W3C XSD binary data type.
    </xs:documentation></xs:annotation>
    </xs:element>
    <xs:element name = "serializable">
     <xs:complexType>
        <xs:attribute name="uid" type="xs:long" default="1"/>
    </xs:complexType>
     </xs:element>
</xs:schema>
```

APPENDIX D

BINDING XML NAMES TO JAVA IDENTIFIERS

D.1 Overview

This section provides default mappings from:

- XML Name to Java identifier
- Model group to Java identifier
- Namespace URI to Java package name

D.2 The Name to Identifier Mapping Algorithm

Java identifiers typically follow three simple, well-known conventions:

- Class and interface names always begin with an upper-case letter. The remaining characters are either digits, lower-case letters, or upper-case letters. Upper-case letters within a multi-word name serve to identify the start of each non-initial word, or sometimes to stand for acronyms.
- Method names and components of a package name always begin with a lower-case letter, and otherwise are exactly like class and interface names.
- Constant names are entirely in upper case, with each pair of words separated by the underscore character ('_', \u005F, LOW LINE).

XML names, however, are much richer than Java identifiers: They may include not only the standard Java identifier characters but also various punctuation and special characters that are not permitted in Java identifiers. Like most Java identifiers, most XML names are in practice composed of more than one natural-language word. Non-initial words within an XML name typically start with an upper-case letter followed by a lower-case letter, as in Java language, or are prefixed by punctuation characters, which is not usual in the Java language and, for most punctuation characters, is in fact illegal.

In order to map an arbitrary XML name into a Java class, method, or constant identifier, the XML name is first broken into a *word list*. For the purpose of constructing word lists from XML names we use the following definitions:

- A *punctuation character* is one of the following:
  - A hyphen ('-', \u002D, HYPHEN-MINUS),
  - A period ('.', \u002E, FULL STOP),
  - A colon (':', \u003A, COLON),
  - A dot ('·', \u00B7, MIDDLE DOT),
  - \u0387, GREEK ANO TELEIA,
  - \u06DD, ARABIC END OF AYAH, or
  - \u06DE, ARABIC START OF RUB EL HIZB.
  - An underscore ('_', \u005F, LOW LINE) with following exception[1]

These are all legal characters in XML names.

- A *letter* is a character for which the `Character.isLetter` method returns `true`, *i.e.*, a letter according to the Unicode standard. Every letter is a legal Java identifier character, both initial and non-initial.
- A *digit* is a character for which the `Character.isDigit` method returns `true`, *i.e.*, a digit according to the Unicode Standard. Every digit is a legal non-initial Java identifier character.
- A *mark* is a character that is in none of the previous categories but for which the `Character.isJavaIdentifierPart` method returns `true`. This category includes numeric letters, combining marks, non-spacing marks, and ignorable control characters.

---

1. Exception case: Underscore is not considered a punctuation mark for schema customization `<jaxb:globalBindings underscoreHandling="asCharInWord"/>` specified in Section 7.5.3, "Underscore Handling". For this customization, underscore is considered a special letter that never results in a word break as defined in Table D-1 and it is definitely not considered an uncased letter. See example bindings in Table D-3.

Every XML name character falls into one of the above categories. We further divide letters into three subcategories:

- An *upper-case letter* is a letter for which the `Character.isUpperCase` method returns `true`,
- A *lowercase letter* is a letter for which the `Character.isLowerCase` method returns `true`, and
- All other letters are *uncased*.

An XML name is split into a word list by removing any leading and trailing punctuation characters and then searching for *word breaks*. A word break is defined by three regular expressions: A prefix, a separator, and a suffix. The prefix matches part of the word that precedes the break, the separator is not part of any word, and the suffix matches part of the word that follows the break. The word breaks are defined as:

Table 4-1 XML Word Breaks

Prefix	Separator	Suffix	Example
`[^punct]`	`punct+`[1]	`[^punct]`	`foo|--|bar`
`digit`		`[^digit]`	`foo|22|bar`
`[^digit]`		`digit`	`foo|22`
`lower`		`[^lower]`	`foo|Bar`
`upper`		`upper lower`	`FOO|Bar`
`letter`		`[^letter]`	`Foo|\u2160`
`[^letter]`		`letter`	`\u2160|Foo`
`uncased`		`[^uncased]`	
`[^uncased]`		`uncased`	

(The character `\u2160` is ROMAN NUMERAL ONE, a numeric letter.)

After splitting, if a word begins with a lower-case character then its first character is converted to upper case. The final result is a word list in which each word is either

- A string of upper- and lower-case letters, the first character of which is upper case (includes underscore,'_', for exception case[1]).
- A string of digits, or
- A string of uncased letters and marks.

Given an XML name in word-list form, each of the three types of Java identifiers is constructed as follows:

- A class or interface identifier is constructed by concatenating the words in the list,
- A method identifier is constructed by concatenating the words in the list. A prefix verb (`get`, `set`, *etc.*) is prepended to the result.
- A constant identifier is constructed by converting each word in the list to upper case; the words are then concatenated, separated by underscores.

This algorithm will not change an XML name that is already a legal and conventional Java class, method, or constant identifier, except perhaps to add an initial verb in the case of a property access method.

To improve user experience with default binding, the automated resolution of frequent naming collision is specified in Section D.2.1.1, “Standardized Name Collision Resolution“.

Example

Table 4-2 XML Names and derived Java Class, Method, and Constant Names

XML Name	Class Name	Method Name	Constant Name
mixedCaseName	MixedCaseName	getMixedCaseName	MIXED_CASE_NAME
Answer42	Answer42	getAnswer42	ANSWER_42
name-with-dashes	NameWithDashes	getNameWithDashes	NAME_WITH_DASHES
other_punct-chars	OtherPunctChars	getOtherPunctChars	OTHER_PUNCT_CHARS

Table 4-3 XML Names and derived Java Class, Method, and Constant Names when <jaxb:globalBindings underscoreHandling=”asCharInWord”>

XML Name	Class Name	Method Name	Constant Name
other_punct-chars	Other_punctChars	getOther_punctChars	OTHER_PUNCT_CHARS
name_with_underscore	Name_with_underscore	name_with_underscore	NAME_WITH_UNDERSCORE

D.2.1 Collisions and conflicts

It is possible that the name-mapping algorithm will map two distinct XML names to the same word list.These cases will result in a *collision* if, and only if, the same Java identifier is constructed from the word list and is used to name two distinct generated classes or two distinct methods or constants in the same

generated class. It is also possible if two or more namespaces are customized to map to the same Java package, XML names that are unique due to belonging to distinct namespaces could mapped to the same Java Class identifier. Collisions are not permitted by the schema compiler and are reported as errors; they may be repaired by revising XML name within the source schema or by specifying a customized binding that maps one of the two XML names to an alternative Java identifier.

A class name must not conflict with the generated JAXB class, `ObjectFactory`, section 5.2 on page 50, that occurs in each schema-derived Java package. Method names are forbidden to conflict with Java keywords or literals, with methods declared in `java.lang.Object`, or with methods declared in the binding-framework classes. Such conflicts are reported as errors and may be repaired by revising the appropriate schema or by specifying an appropriate customized binding that resolves the name collision.

D.2.1.1 Standardized Name Collision Resolution

Given the frequency of an XML element or attribute with the name “class” or “Class” resulting in a naming collision with the inherited method `java.lang.Object.getClass()`, method name mapping automatically resolves this conflict by mapping these XML names to the java method identifier “getClazz”.

Design Note – The likelihood of collisions, and the difficulty of working around them when they occur, depends upon the source schema, the schema language in which it is written, and the binding declarations. In general, however, we expect that the combination of the identifier-construction rules given above, together with good schema-design practices, will make collisions relatively uncommon.

The capitalization conventions embodied in the identifier-construction rules will tend to reduce collisions as long as names with shared mappings are used in schema constructs that map to distinct sorts of Java constructs. An attribute named `foo` is unlikely to collide with an element type named `foo` because the first maps to a set of property access methods (`getFoo`, `setFoo`, *etc.*) while the second maps to a class name (`Foo`).

Good schema-design practices also make collisions less likely. When writing a schema it is inadvisable to use, in identical roles, names that are distinguished only by punctuation or case. Suppose a schema declares two attributes of a single element type, one named `Foo` and the other named `foo`. Their generated access methods, namely `getFoo` and `setFoo`, will collide. This situation would best be handled by revising the source schema, which would not only eliminate the collision but also improve the readability of the source schema and documents that use it.

D.3 Deriving a legal Java identifier from an enum facet value

Given that an enum facet's value is not restricted to an XML name, the XML Name to Java identifier algorithm is not applicable to generating a Java identifier from an enum facet's value. The following algorithm maps an enum facet value to a valid Java constant identifier name.

- For each character in enum facet value,
  copy the character to a string representation `javaId` when `java.lang.Character.isJavaIdentifierPart()` is `true`.
  - To follow Java constant naming convention, each valid lower case character must be copied as its upper case equivalent.
- There is no derived Java constant identifier when any of the following occur:

- `javaId.length() == 0`
- `java.lang.Character.isJavaIdentifierStart(javaId.get(0)) == false`

D.4 Deriving an identifier for a model group

XML Schema has the concept of a group of element declarations. Occasionally, it is convenient to bind the grouping as a Java content property or a Java value class. When a semantically meaningful name for the group is not provided within the source schema or via a binding declaration customization, it is necessary to generate a Java identifier from the grouping. Below is an algorithm to generate such an identifier.

A name is computed for an unnamed model group by concatenating together the first 3 element declarations and/or wildcards that occur within the model group. Each XML *{name}* is mapped to a Java identifier for a method using the XML Name to Java Identifier Mapping algorithm. Since wildcard does not have a {name} property, it is represented as the Java identifier "`Any`". The Java identifiers are concatenated together with the separator "`And`" for sequence and all compositor and "`Or`" for choice compositors. For example, a sequence of element `foo` and element `bar` would map to "`FooAndBar`" and a choice of element `foo` and element `bar` maps to "`FooOrBar`." Lastly, a sequence of wildcard and element `bar` would map to the Java identifier "`AnyAndBar`".

Example:

Given XML Schema fragment:

```
<xs:choice>
    <xs:sequence>
        <xs:element ref="A"/>
        <xs:any processContents="strict"/>
    </xs:sequence>
    <xs:element ref="C"/>
</xs:choice>
```

The generated Java identifier would be `AAndAnyOrC`.

D.5 Generating a Java package name

This section describes how to generate a package name to hold the derived Java representation. The motivation for specifying a default means to generate a Java package name is to increase the chances that a schema can be processed by a schema compiler without requiring the user to specify customizations.

If a schema has a target namespace, the next subsection describes how to map the URI into a Java package name. If the schema has no target namespace, there is a section that describes an algorithm to generate a Java package name from the schema filename.

D.5.1 Mapping from a Namespace URI

An XML namespace is represented by a URI. Since XML Namespace will be mapped to a Java package, it is necessary to specify a default mapping from a URI to a Java package name. The URI format is described in [RFC2396].

The following steps describe how to map a URI to a Java package name. The example URI, `http://www.acme.com/go/espeak.xsd`, is used to illustrate each step.

1. Remove the scheme and `":"` part from the beginning of the URI, if present.
   Since there is no formal syntax to identify the optional URI scheme, restrict the schemes to be removed to case insensitive checks for schemes "`http`" and "`urn`".

   ```
   //www.acme.com/go/espeak.xsd
   ```

2. **Remove the trailing file type, one of `.??` or `.???` or `.html`.**

   ```
   //www.acme.com/go/espeak
   ```

3. **Parse the remaining string into a list of strings using `'/'` and `':'` as separators. Treat consecutive separators as a single separator.**

   ```
   {"www.acme.com", "go", "espeak" }
   ```

4. **For each string in the list produced by previous step, unescape each escape sequence octet.**

   ```
   {"www.acme.com", "go", "espeak" }
   ```

5. **If the scheme is a "urn", replace all dashes, "-", occurring in the first component with ".".**[2]

6. **Apply algorithm described in Section 7.7 "Unique Package Names" in [JLS] to derive a unique package name from the potential internet domain name contained within the first component. The internet domain name is reversed, component by component. Note that a leading "`www`." is not considered part of an internet domain name and must be dropped.**

   If the first component does not contain either one of the top-level domain names, for example, com, gov, net, org, edu, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981, this step must be skipped.

   ```
   {"com", "acme", "go", "espeak"}
   ```

7. **For each string in the list, convert each string to be all lower case.**

   ```
   {"com", "acme", "go", "espeak" }
   ```

8. **For each string remaining, the following conventions are adopted from [JLS] Section 7.7, "Unique Package Names."**

   a. If the sting component contains a hyphen, or any other special character not allowed in an identifier, convert it into an underscore.

   b. If any of the resulting package name components are keywords then append underscore to them.

   c. If any of the resulting package name components start with a digit, or any other character that is not allowed as an initial character of an identifier, have an underscore prefixed to the component.

   ```
   {"com", "acme", "go", "espeak" }
   ```

9. **Concatenate the resultant list of strings using '.' as a separating character to produce a package name.**

   ```
   Final package name: "com.acme.go.espeak".
   ```

Section D.2.1, "Collisions and conflicts," on page 336, specifies what to do when the above algorithm results in an invalid Java package name.

[2] Sample URN "urn:hl7-org:v3" {"h17-org", "v3") transforms to {"h17.org", "v3"}.

D.6 Conforming Java Identifier Algorithm

This section describes how to convert a legal Java identifier which may not conform to Java naming conventions to a Java identifier that conforms to the standard naming conventions. Section 7.5.2, “Customized Name Mapping“discusses when this algorithm is applied to customization names.

Since a legal Java identifier is also a XML name, this algorithm is the same as Section D.2, “The Name to Identifier Mapping Algorithm” with the following exception: constant names must not be mapped to a Java constant that conforms to the Java naming convention for a constant.

APPENDIX **E**

EXTERNAL BINDING DECLARATION

E.1 Example

Example: Consider the following schema and external binding file:

Source Schema: `A.xsd:`

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:ens="http://example.com/ns"
           targetNamespace="http://example.com/ns">
    <xs:complexType name="aType">
        <xs:sequence>
            <xs:element name="foo" type="xs:int"/>
        </xs:sequence>
        <xs:attribute name="bar" type="xs:int"/>
    </xs:complexType>
    <xs:element name="root" type="ens:aType"/>
</xs:schema>
```

External binding declarations file:

```
<jaxb:bindingsxmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xs="http://www.w3.org/2001/XMLSchema"
               version="1.0">
    <jaxb:bindings schemaLocation="A.xsd">
        <jaxb:bindings node="//xs:complexType[@name='aType']">
            <jaxb:class name="customNameType"/>
            <jaxb:bindings node=".//xs:element[@name='foo']">
                <jaxb:property name="customFoo"/>
            </jaxb:bindings>
```

```
            <jaxb:bindings node="./xs:attribute[@name='bar']">
                <jaxb:property name="customBar"/>
            </jaxb:bindings>
        </jaxb:bindings>
    </jaxb:bindings>
</jaxb:bindings>
```

Conceptually, the combination of the source schema and external binding file above are the equivalent of the following inline annotated schema.

```
<xs:schema      xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:ens="http://example.com/ns"
                targetNamespace="http://example.com/ns"
                xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
                jaxb:version="1.0">
    <xs:complexType name="aType">
        <xs:annotation>
            <xs:appinfo>
                <jaxb:class name="customNameType"/>
            </xs:appinfo>
        </xs:annotation>
        <xs:sequence>
            <xs:element name="foo" type="xs:int">
                <xs:annotation>
                    <xs:appinfo>
                        <jaxb:property name="customFoo"/>
                    </xs:appinfo>
                </xs:annotation>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="bar" type="xs:int">
            <xs:annotation>
                <xs:appinfo>
                    <jaxb:property name="customBar"/>
                </xs:appinfo>
            </xs:annotation>
        </xs:attribute>
    </xs:complexType>
    <xs:element name="root" type="ens:aType"/>
</xs:schema>
```

E.2 Transformation

The intent of this section is to describe the transformation of external binding declarations and their target schemas into a set of schemas annotated with JAXB binding declarations. ready for processing by a JAXB compliant schema compiler.

This transformation must be understood to work on XML data model level. Thus, this transformation is applicable even for those schemas which contain semantic errors.

The transformation is applied as follows:

1. Gather all the top-most <jaxb:bindings> elements from all the schema documents and all the external binding files that participate in this process. *Top-most* <jaxb:bindings> are those <jaxb:bindings> elements that are either a root element in a document or whose parent is an <xs:appinfo> element. We will refer to these trees as "external binding forest."
2. **Collect all the namespaces used in the elements inside the external binding forest, except the taxi namespace, "http://java.sun.com/xml/ns/jaxb", and the no namespace. Allocate an unique prefix for each of them and declare the namespace binding at all the root <xs:schema> elements of each schema documents. Then add a jaxb:extensionBindingPrefix attribute to each <xs:schema> element with all those allocated prefixes. If an <xs:schema> element already carries this attribute, prefixes are just appended to the existing attributes.**

   Note: The net effect is that all "foreign" namespaces used in the external binding forest will be automatically be considered as extension customization declaration namespaces.
3. **For each <jaxb:bindings> element, we determine the "target element" that the binding declaration should be associated with. This process proceeds in a top-down fashion as follows:**

   a. Let p be the target element of the parent <jaxb:bindings>. If it is the top most <jaxb:bindings>, then let p be the <jaxb:bindings> element itself.

b. Identify the "target element" using <jaxb:bindings> attributes.
(i) If the <jaxb:bindings> has a @schemaLocation, the value of the attribute should be taken as an URI and be absolutized with the base URI of the <jaxb:bindings> element. Then the target element will be the root node of the schema document identified by the absolutized URI. If there's no such schema document in the current input, it is an error. Note: the root node of the schema document is not the document element.

(ii) If the <jaxb:bindings> has @node, the value of the attribute should be evaluated as an XPath 1.0 expression. The context node in this evaluation should be p as we computed in the previous step. It is an error if this evaluation results in something other than a node set that contains exactly one element. Then the target element will be this element.

(iii) if the <jaxb:bindings> has neither @schemaLocation nor @node, then the target element will be p as we computed in the previous step. Note: <jaxb:bindings> elements can't have both @schemaLocation and @node at the same time.

We define the target element of a binding declaration to be the target element of its parent <jaxb:bindings> element. The only exception to this is <jaxb:globalBindings> binding declaraiton, in which case the target element will be the document element of any one of the schema documents being compiled (such choice is undeterministic, but the semantics of <jaxb:globalBindings> is not affected by this choice, so the end result will be the same.) It is an error if a target element of a binding declaration doesn't belong to the *"http://wwww.w3.org/2001/XMLSchema"* namespace.

4. **Next, for each target element of binding declarations, if it doesn't have any <xs:annotation> <xs:appinfo> in its children, one will be created and added as the first child of the target.**

   After that, we move each binding declaration under the target node of its parent <jaxb:bindings>. Consider the first <xs:appinfo> child of the target element. The binding declaration element will be moved under this <xs:appinfo> element.

APPENDIX **F**

XML SCHEMA

F.1 Abstract Schema Model

The following summarization abstract schema component model has been extracted from [XSD Part 1] as a convenience for those not familiar with XML Schema component model in understanding the binding of XML Schema components to Java representation. One must refer to [XSD Part 1] for the complete normative description for these components.

F.1.1 Simple Type Definition Schema Component

Table 6-1 Simple Type Definition Schema Components

Component	Description
`{name}`	Optional. An NCName as defined by [XML-Namespaces].
`{target namespace}`	Either ·absent· or a namespace name.
`{base type definition}`	A simple type definition
`{facets}`	A set of constraining facets.
`{fundamental facets}`	A set of fundamental facets.
`{final}`	A subset of {extension, list, restriction, union}.

Table 6-1 Simple Type Definition Schema Components *(Continued)*

Component	Description	
{variety}	One of {atomic, list, union}. Depending on the value of {variety}, further properties are defined as follows:	
	atomic {primitive type definition}	A built-in primitive simple type definition.
	list {item type definition}	A simple type definition.
	union {member type definitions}	A non-empty sequence of simple type definitions.
{annotation}	Optional. An annotation.	

F.1.2 Enumeration Facet Schema Component

Table 6-2 Enumeration Facet Schema Components

Component	Description
{value}	The actual value of the value. (Must be in value space of base type definition.)
{annotation}	Optional annotation.

F.1.3 Complex Type Definition Schema Component

Table 6-3 Complex Type Definition Schema Components

Component	Description
{name}	Optional. An NCName as defined by [XML-Namespaces].
{target namespace}	Either ·absent· or a namespace name.
{base type definition}	Either a simple type definition or a complex type definition.
{scope}	Either *global* or a complex type definition
{derivation method}	Either extension or *restriction*.
{final}	A subset of {extension, restriction}.
{abstract}	A boolean
{attribute uses}	A set of attribute uses.

Table 6-3 Complex Type Definition Schema Components *(Continued)*

Component	Description
{attribute wildcard}	Optional. A wildcard.
{content type}	One of *empty*, a *simple type definition*, or a pair consisting of a ·content model· and one of *mixed*, *element-only*.
{prohibited substitutions}	A subset of {extension, restriction}.
{substitution group affiliation}	Optional. If exists, this element declaration belongs to a substitution group and this specified element name is the QName of the substitution head.
{annotations}	A set of annotations.

F.1.4 Element Declaration Schema Component

Table 6-4 Element Declaration Schema Components

Component	Description
{name}	An NCName as defined by [XML-Namespaces].
{target namespace}	Either ·absent· or a namespace name
{type definition}	Either a simple type definition or a complex type definition.
{scope}	Optional. Either global or a complex type definition.
{value constraint}	Optional. A pair consisting of a value and one of default, fixed.
{nillable}	A boolean.
{identity-constraint definitions}	A set of constraint definitions.
{substitution group affiliation}	Optional. A top-level element definition.
{substitution group exclusions}	A subset of {extension, restriction}.
{disallowed substitution}	A subset of {substitution,extension,restriction}.
{abstract}	A boolean.
{annotation}	Optional. An annotation.

F.1.5 Attribute Declaration Schema Component

Table 6-5 Attribute Declaration Schema Components

Component	Description
{name}	An NCName as defined by [XML-Namespaces].
{target namespace}	If form is present and is "qualified", or if form is absent and the value of @attributeFormDefault on the <schema> ancestor is "qualified", then the schema's {targetNamespace}, or ·absent· if there is none, otherwise ·absent·
{type definition}	A simple type definition.
{scope}	Optional. Either global or a complex type definition.
{value constraint}	Optional. A pair consisting of a value and one of default, fixed.
{annotation}	Optional. An annotation.

F.1.6 Model Group Definition Schema Component

Table 6-6 Model Group Definition Schema Components

Component	Description
{name}	An NCName as defined by [XML-Namespaces].
{target namespace}	Either ·absent· or a namespace name.
{model group}	A model group.
{annotation}	Optional. An annotation.

F.1.7 Identity-constraint Definition Schema Component

Table 6-7 Identity-constraint Definition Schema Components

Component	Description
{name}	An NCName as defined by [XML-Namespaces].
{target namespace}	Either ·absent· or a namespace name.
{identity-constraint category}	One of key, keyref or unique.
{selector}	A restricted XPath ([XPath]) expression.

Table 6-7 Identity-constraint Definition Schema Components *(Continued)*

Component	Description
{fields}	A non-empty list of restricted XPath ([XPath]) expressions.
{referenced key}	Required if {identity-constraint category} is keyref, forbidden otherwise. An identity-constraint definition with {identity-constraint category} equal to key or unique.
{annotation}	Optional. An annotation.

F.1.8 Attribute Use Schema Component

Table 6-8 Attribute Use Schema Components

Component	Description
{required}	A boolean.
{attribute declaration}	An attribute declaration.
{value constraint}	Optional. A pair consisting of a value and one of default, fixed.

F.1.9 Particle Schema Component

Table 6-9 Particle Schema Components

Component	Description
{min occurs}	A non-negative integer.
{max occurs}	Either a non-negative integer or unbounded.
{term}	One of a model group, a wildcard, or an element declaration.

F.1.10 Wildcard Schema Component

Table 6-10 Wildcard Schema Components

Component	Description
{namespace constraint}	One of any; a pair of not and a namespace name or ·absent·; or a set whose members are either namespace names or ·absent·.

Table 6-10 Wildcard Schema Components

Component	Description
`{process contents}`	One of skip, lax or strict.
`{annotation}`	Optional. An annotation.

F.1.11 Model Group Schema Component

Table 6-11 Model Group Components

Component	Description
`{compositor}`	One of *all*, *choice* or *sequence*.
`{particles}`	A list of particles.
`{annotation}`	An annotation.

F.1.12 Notation Declaration Schema Component

Table 6-12 Notation Declaration Components

Component	Description
`{name}`	An NCName as defined by [XML-Namespaces].
`{target namespace}`	Actual value of the targetNamespace [attribute] of the parent schema element
`{system identifier}`	The ·actual value· of the system [attribute], if present, otherwise absent.
`{public identifier}`	The ·actual value· of the public [attribute]
`{annotation}`	Optional. An annotation.

F.1.13 Wildcard Schema Component

Table 6-13 Wildcard Components

Component	Description
`{namespace constraint}`	One of `any`; a pair of `not` and a namespace name or ·absent·; or a set whose members are either namespace names or ·absent·.
`{process contents}`	One of `skip`, `lax` or `strict`.
`{annotation}`	Optional. An annotation.

F.1.14 Attribute Group Definition Schema Component

Table 6-14 Attribute Group Definition Schema Components

Component	Description
`{name}`	An NCName as defined by [XML-Namespaces].
`{target namespace}`	Either ·absent· or a namespace name.
`{attribute uses}`	A set of attribute uses.
`{attribute wildcard}`	Optional. A wildcard. *(part of the complete wildcard)*
`{annotation}`	

APPENDIX G

DEPRECATED JAXB 1.0 FUNCTIONALITY

G.1 Overview

Due to significant re-architecture in JAXB 2.0 to improve efficiency, usability, schema-derived footprint and binding framework runtime footprint, certain JAXB 1.0 operations are no longer required to be implemented in JAXB 2.0. These deprecated operations do still need to be supported by a JAXB 2.0 binding runtime for JAXB 1.0 schema-derived classes.

G.2 On-demand Validation

It is optional for a JAXB 2.0 implementation to implement on-demand validation for JAXB 2.0 mapped classes. There is no reasonable way to implement this functionality of JAXB 2.0 annotated classes and leverage JAXP 1.3 validation facility. For backwards compatibility, an implementation is required to support this functionality for JAXB 1.0 schema-derived classes.

G.2.1 Validator for JAXB 1.0 schema-derived classes

The following text is from JAXB 1.0 Specification and applies to JAXB 1.0 schema-derived classes.

An application may wish to validate the correctness of the Java content tree based on schema validation constraints. This form of validation enables an

application to initiate the validation process on a Java content tree at a point in time that it feels it should be valid. The application is notified about validation errors and warnings detected in the Java content tree.

The `Validator` class is responsible for controlling the validation of a content tree of in-memory objects. The following summarizes the available operations on the class.

```
public interface Validator {
    ValidationEventHandler getEventHandler()
    void setEventHandler(ValidationEventHandler)

    boolean validate(java.lang.Object subrootObject)
    boolean validateRoot(java.lang.Object rootObject)

    java.lang.Object getProperty(java.lang.String name)
    void setProperty(java.lang.String name, java.lang.Object value)
}
```

`The JAXBContext` class provides a factory to create a `Validator` instance. After an application has made a series of modifications to a Java content tree, the application validates the content tree on-demand. As far as the application is concerned, this validation takes place against the Java content instances and validation constraint warnings and errors are reported to the application relative to the Java content tree. Validation is initiated by invoking the `validateRoot(Object)` method on the root of the Java content tree or by invoking `validate(Object)` method to validate any arbitrary subtree of the Java content tree. The only difference between these two methods is global constraint checking (i.e. verifying ID/IDREF constraints.) The `validateRoot(Object)` method includes global constraint checking as part of its operation, whereas the `validate(Object)` method does not.

The validator governs the process of validating the content tree, serves as a registry for identifier references, and ensures that all local (and when appropriate, global) structural constraints are checked before the validation process is complete.

If a violation of a local or global structural constraint is detected, then the application is notified of the event with a callback passing an instance of a `ValidationEvent` as a parameter.

APPENDIX **H**

ENHANCED BINARY DATA HANDLING

H.1 Overview

Optimized transmission of binary data as attachments is described by standards such as Soap [MTOM]/Xml-binary Optimized Packaging[XOP] and WS-I Attachment Profile 1.0 [WSIAP]. To optimally support these standards when JAXB databinding is used within a message passing environment, Section H.3, "javax.xml.bind.attachments"specifies an API that allows for an integrated, cooperative implementation of these standards between a MIME-based package processor and the JAXB 2.0 unmarshal/marshal processes. An enhanced binding of MIME content to Java representation is specified in Section H.2, "Binding MIME Binary Data".

H.2 Binding MIME Binary Data

H.2.1 Binary Data Schema Annotation

As specified in [MIME], the XML Schema annotation attribute, `xmime:expectedContentTypes`, lists the expected MIME content-type(s) for element content whose type derives from the xsd binary datatypes,`xs:base64Binary` or `xs:hexBinary`.

JAXB 2.0 databinding recognizes this schema constraint to improve the binding of MIME type constrained binary data to Java representation. The `xmime:expectedContentType` attribute is allowed on type definitions deriving from binary datatypes and on element declarations with types that

derive from binary datatypes. For JAXB 2.0 binding purposes, the schema annotation, `xmime:expectedContentTypes` is evaluated for binding purposes for all cases EXCEPT when the annotation is on an element declaration with a named complex type definition. For that case, the `xmime:expectedContentTypes` annotation must be located directly within the scope of the complex type definition in order to impact the binding of the complex type definition's simple binary content.

H.2.1.1 Binding Known Media Type

When `@xmime:expectedContentTypes` schema annotation only refers to one MIME type, it is considered a known media type for the binary data. [MIME] does not require an `xmime:contentType` attribute to hold the dynamic mime type for the binary data for this case. JAXB binding can achieve an optimal binding for this case. The default MIME type to Java datatype are in Table H-1.

Table H-1 Default Binding for Known Media Type

MIME Type	Java Type
image/gif	java.awt.Image
image/jpeg	java.awt.Image
text/xml or application/xml	javax.xml.transform.Source
any other MIME types	javax.activation.DataHandler

A JAXB program annotation element, `@XmlMimeType`, is generated to preserve the known media type for use at marshal time.

CODE EXAMPLE 8-1 schema with a known media type

```
<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:tns="http://example.com/know-type"
|            xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
           targetNamespace="http://example.com/know-type">
    <xs:import namespace="http://www.w3.org/2005/05/xmlmime"
            schemaLocation="http://www.w3.org/2005/05/xmlmime"/>
    <xs:element name="JPEGPicture" type="xs:base64binary"
                      xmime:expectedContentTypes="image/jpeg"/>
</xs:schema>
```

CODE EXAMPLE 8-2 JAXB 2.0 binding of Example 9-1

```
import java.awt.Image;
@XmlRegistry
class ObjectFactory {
    @XmlElementDecl(...)
    @XmlMimeType("image/jpeg")
    JAXBELement<Image> createJPEGPicture(Image value);
}
```

The `@XmlMimeType` annotation provides the MIME content type needed by JAXB 2.0 Marshaller to specify the mime type to set `DataHandler.setContentType(String)`.

CODE EXAMPLE 8-3 Schema for local element declaration annotated with known media type

```
<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="http://example.com/know-type"
           xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
           targetNamespace="http://example.com/know-type">
    <xs:import namespace="http://www.w3.org/2005/05/xmlmime"
           schemaLocation="http://www.w3.org/2005/05/xmlmime"/>
    <xs:complexType name="Item">
        <xs:sequence>
            <xs:element name="JPEGPicture"
                        type="xs:base64Binary"
                        xmime:expectedContentTypes="image/jpeg"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

CODE EXAMPLE 8-4 Java Binding of Section CODE EXAMPLE H-3, “Schema for local element declaration annotated with known media type“

```
import java.awt.Image;
public class Item {
    @XmlMimeType("image/jpeg")
    Image getJPEGPicture();
    void setJPEGPicture(Image value);
}
```

H.2.1.2 Binding Preferred Media Types

If there are more than one mime type listed in `xmime:expectedContentTypes` or if there is one with a wildcard in it, the annotation specifies the Preferred Media Types and recommends that the

binary data be simple content that has an attribute `xmime:contentType` that specifies which of the `xmime:expectedContentTypes` the binary data represents.

Given that the exact media type is not known for this case, a Preferred Media Type binds to `javax.activation.DataHandler`. `DataHandler` has a property `get/setContentType` that should be kept synchronized with the value of the JAXB binding for the `xmime:contentType` attribute.

H.2.2 Binding WS-I Attachment Profile `ref:swaRef`

An XML element or attribute with a type definition of `ref:swaRef` is bound to a JAXB property with base type of `javax.activation.DataHandler` and annotated with `@XmlAttachmentRef`.

H.3 javax.xml.bind.attachments

The abstract classes `AttachmentUnmarshaller` and `AttachmentMarshaller` in package `javax.xml.bind.attachments` are intended to be implemented by a MIME-based package processor, such as JAX-WS 2.0 implementation, and are called during JAXB unmarshal/marshal. The JAXB unmarshal/marshal processes the root part of a MIME-based package, delegating knowledge of the overall package and its other parts to the `Attachment*` class implementations.

H.3.1 AttachmentUnmarshaller

An implementation of this abstract class by a MIME-based package processor provides access to package-level information that is outside the scope of the JAXB unmarshal process. A MIME-based package processor registers its processing context with a JAXB 2.0 processor using the method `setAttachmentUnmarshaller(AttachmentUnmarshaller)` of `javax.xml.bind.Unmarshaller`.

Interactions between the Unmarshaller and the abstract class are summarized below. The javadoc specifies the details.

```
public abstract class AttachmentUnmarshaller {
    public boolean isXOPPackage();
    public abstract DataHandler getAttachmentAsDataHandler(String cid);
    public abstract byte[] getAttachmentAsByteArray(String cid);
}
```

The JAXB unmarshal process communicates with a MIME-based package processor via an instance of AttachmentUnmarshaller registered with the unmarshaller. Figure H.1 summarizes this processing.

- MTOM/XOP processing during unmarshal:
  When `isXOPPackage()` returns true, the unmarshaller replaces each XOP include element it encounters with MIME content returned by the appropriate `getAttachment*()` method.
- WS-I AP processing:
  Each element or attribute of type definition `ref:swaRef`, a content-id uri reference to binary data, is resolved by the unmarshal process by a call to the appropriate `getAttachment*()` method.

H.3.2 AttachmentMarshaller

An `AttachmentMarshaller` instance is registered with a `javax.xml.bind.Marshaller` instance using the method `Marshaller.setAttachmentMarshaller()`.

Interactions between the Marshaller and the abstract class is summarized below. See the javadoc for details.

```
public abstract class AttachmentMarshaller {
    public boolean isXOPPackage();
    public abstract String
           addMtomAttachment(DataHandler data,
                             String elementNamespace,
                             String elementLocalName);
    public abstract String
           addMtomAttachment(byte[] data,
                             String elementNamespace,
                             String elementLocalName);
    public abstract String addSwaRefAttachment(DataHandler data);
}
```

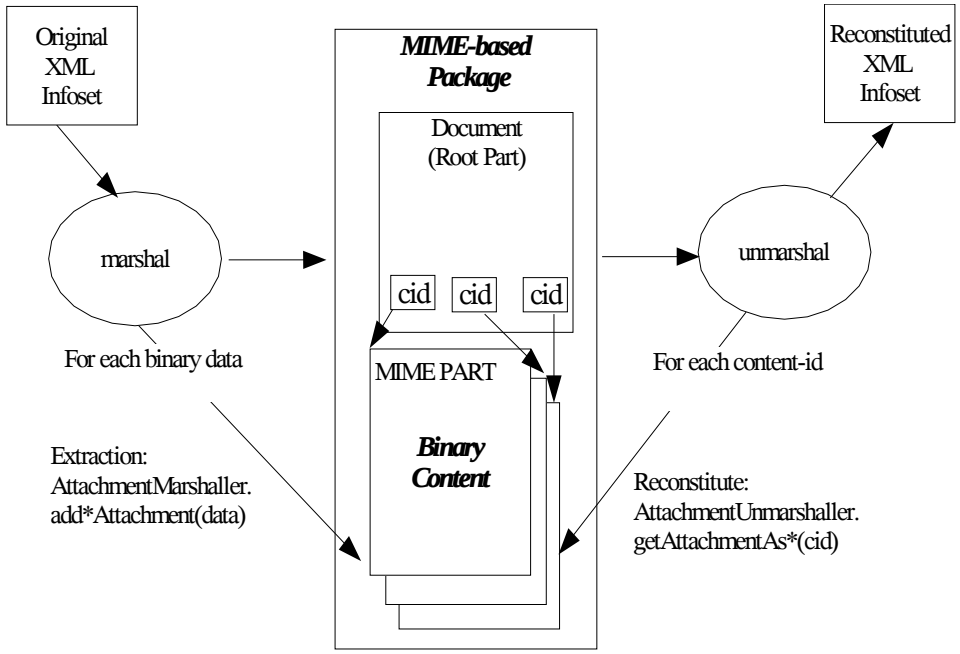
When an AttachmentMarshaller instance is registered with the Marshaller, the following processing takes place.

- MTOM/XOP processing:
  When `isXOPPackage`() is true and a JAXB property representing binary data is being marshalled, the method `addMtomAttachment`(...) is called to provide the MIME-based package processor the opportunity to decide to optimize or inline the binary data.

  Note that the schema customization specified in Section 7.13, "`<inlineBinaryData>` Declaration"can be used to declaratively disable XOP processing for binary data.
- WS-I AP processing:
  The `addSwaRefAttachment` method is called when marshalling content represented by a `ref:swaRef` type definition.

  One can declaratively customize swaRef processing within a schema using schema customization @attachmentRef of <jaxb:property>, specified in Section 7.8.1, "Usage".

Figure H.1 JAXB marshal/unmarshalling of optimized binary content.



APPENDIX I

CHANGE LOG

I.1 Changes since Proposed Final Draft

- Section 7.6.1.2, nameXmlTransform: Apply customization [`jaxb:nameXmlTransform`] addition of prefix and/or suffix *after* XML to Java name transformation is applied.
- Section 6.7.1-2 changed to allow generation of element factory method for abstract element. Change was necessary to support element substitution. The abstract element factory method is generated so it can be annotated with JAXB program annotation that enables element substitution, `@XmlElementDecl.substitutionHeadName`.
- Section 7.7.3.5 fixed the example with <class> customization. Made the corresponding change in Section 6.7.2 so Objectfactory method creates an instance of generated class.
- Chapter 8 and appendix B: @XmlJavaTypeAdapter on class, interface or enum type is mutually exclusive with any other annotation.
- Chapter 8: added @XmlElement.required() for schema generation
- Section 8.7.1.2: clarifications for no-arg static factory method in @XmlType annotation.
- Section 8.9.13.2: Disallow use of @XmlList on single valued property.
- Section 8.9.8.2, Table 8-30 : @XmlAnyAttribute maps to anyAttribute with a namespace constraint with ##other.
- Section 8.9.1.2: If @XmlElement.namespace() is different from that of the target namespace of the enclosing class, require a global element to be generated in the namespace specified in @XmlElement.namespace() to make the generated schema complete.
- Section 8.9.15: Allow @XmlMimeType on a parameter.

- Section 8.9.16: Allow @XmlAttachmentRef on a parameter.
- Chapter 8: removed constraint check that namespace() annotation element must be a valid namespace URI from different annotations.
- Chapter 8: Java Persistence and JAXB 2.0 alignment related changes.
  constructor requirement: public or protected no-arg constructor
  @AccessType renamed to @XmlAccessType.
  @AccessorOrder renamed to @XmlAccessOrder.
  @XmlTransient is mutually exclusive with other annotations.
  @A property or field that is transient or marked with @XmlTransient and specified in @XmlType.propOrder is an error.
- Chapter 8: Clarifications for generics - type variables with type bound, bounded wildcards and java.util.Map.
- Section 8.9: reworked constraints on the properties to handle different use cases permitted by JavaBean design pattern.
- Section 8: Take elementFormDefault into account when determining the namespace for @XmlElement and @XmlElementWrapper annotations.
- Section 8: Added missing mapping constraints for @XmlElementWrapper. Also disallow use of @XmlIDREF with @XmlElementWrapper.
- Chapter 9, “Compatibility”: clarified schema generator and schema compiler requirements.
- Section B.2.5: Added marshalling of null value as xsi:nil or empty element based upon @XmlElement.required and @XmlElement.nillable annotation elements.
- Section B.5: Added new section and moved runtime requirements on getters/setters to here.

I.2 Changes since Public Review

- Update Chapter 9, “Compatibility” for JAXB 2.0 technology. Additional requirements added for Java Types to XML binding and the running of JAXB 1.0 application in a JAXB 2.0 environment.
- Added external event callback mechanism, `Unmarshaller.Listener` and `Marshaller.Listener`.

- Added new unmarshal method overloading, unmarshal by declaredType, to `Unmarshaller` and `Binder`. Enables unmarshalling a root element that corresponds with a local element declaration in schema.
- Added Section 6.13, “Modifying Schema-Derived Code” describing use of annotation `@javax.annotation.Generated` to distinguish between generated and user-modified code in schema-derived class.
- Element declaration with anonymous complex type definition binds to `@XmlRootElement` annotated class except for cases in Section 6.7.3.1.
- Removed <jaxb:globalBindings nullsInCollection>. The customization <jaxb:property generateElementProperty=”true”> can achieve same desired result.
- Added clarification that mapping two or more target namespaces to same java package can result in naming collision that should be detected as an error by schema compiler.
- Added <jaxb:factoryMethod> customization to enable the resolution of name collisions between factory methods.
- First parameter to any of the overloaded Marshaller.marshal() methods must be a JAXB element; otherwise, method must throw MarshalException. See updated Marshaller javadoc and Section 4.5, “Marshalling” for details.
- Prepend “_”, not “Original”, to a Java class name representing an XML Schema type definition that has been redefined in Section 6.10, “Redefine”.
- Format for class name in `jaxb.index` file clarified in JAXBConext.newInstance(String) method javadoc.
- Clarifications on @dom customization in Section 7.12..
- Chapter 8: Added support for @XmlJavaTypeAdapter at the package level.
- Chapter 8: Added new annotation @XmlJavaTypeAdapters as a container for defining multiple @XmlJavaTypeAdapters at the package level.
- Chapter 8: Added support for @XmlSchemaType at the package level.
- Chapter 8: Added @XmlSchemaTypes as a container annotation for defining multiple @XmlSchemaType annotations at the package level.

- Chapter 8: added lists of annotations allowed with each annotation.
- Chapter 8: Bug fixes and clarifications related to mapping and mapping constraints.
- Chapter 8: Expanded collection types mapped to java.util.Map and java.util.Collection.
- Appendix B. Incorporate event call backs into unmarshalling process.
- Appendix B: Incorporate into unmarshalling process additional unmarshal methods: Binder.unmarshal(..), unmarshal methods that take a declaredType as a parameter - Binder.unmarshal(..., declaredType) and Unmarshaller.unmarshal(...,declaredType).

I.3 Changes since Early Draft 2

- Simple type substitution support added in Section 6.7.4.2.
- Updates to enum type binding. (Section 7.5.1, 7.5.5, 7.10, Appendix D.3)
- Optimized binary data.(Appendix H) and schema customizations. (Section 7.13 and 7.10.5)
- Clarification for `<jaxb:globalBindings underscoreHandling="asCharInWord">` (Appendix D.2)
- Added Unmarshal and Marshal Callback Events (Section 4.4.1,4.5.1)
- Clarification: xs:ID and xs:IDREF can not bind to an enum type. (Section 6.2.3,7.10.5)
- Added schema customization:
  <jaxb:globalBinding localScoping="nested"|"toplevel"> (Section 7.5.1)
  <jaxb:inlineBinaryData> (Section 7.13)
  <jaxb:property @attachmentRef/> (Section 7.8.1)
- Updated Section 6 and 7 with mapping annotations that are generated on schema-derived JAXB classes/properties/fields.
- Added javax.xml.bind.Binder class to Section 4.8.2.
- Runtime generation of schema from JAXB mapping annotations: JAXBContext.generateSchema().
- Chapter 8: added @XmlList: bind property/field to simple list type

- Chapter 8: added @XmlAnyElement: bind property/field to xs:any
- Chapter 8: added @XmlAnyAttribute - bind property/field to xs:anyAttribute
- Chapter 8. added @XmlMixed - for mixed content
- Chapter 8, added annotations for attachment/MTOM support: @XmlMimeType, @XmlAttachmentRef
- Chapter 8: added @XmlAccessorOrder - to specify default ordering.
- Chapter 8: added @XmlSchemaType mainly for use in mapping XMLGregorianCalendar.
- Chapter 8: map java.lang.Object to xs:anyType
- Chapter 8: added mapping of XMLGregorianCalendar
- Chapter 8: added mapping of generics - type variables, wildcardType
- Chapter 8: added mapping of binary data types.
- Chapter 8: default mappings changed for class, enum type.
- Chapter 8: default mapping of propOrder specified.
- Chapter 8: mapping of classes - zero arg constructor, factory method.
- Chapter 8: added Runtime schema generation requirement.
- Chapter 8: Clarified mapping constraints and other bug fixes.
- Added Appendix B new: Added Runtime Processing Model to specify the marshalling/unmarshalling for dealing with invalid XML content and schema evolution.
- Updated Appendix C to JAXB 2.0 binding schema.

I.4 Changes since Early Draft

- Updated goals in Introduction.
- Update to Section 3 “Architecture” introducing Java to Schema binding.
  - section on portable annotation-driven architecture.
  - section on handling of invalid XML content
- Binding Framework

- Replaced `IXmlElement<T>` interface with `JAXBElement<T>` class. (JAXBElement is used for schema to java binding)
- `JAXBIntrospector` introduced.
- Add flexible (by-name) unmarshal and describe JAXB 1.0 structural unmarshalling.
- Moved deprecated on-demand validation, accessible via javax.xml.bind.Validator, to Appendix H.

- XSD to Java Binding
  - Bind complex type definition to value class by default.
  - Schema-derived code is annotated with JAXB java annotations.
  - Bind XSD simpleType with enum facet to J2SE 5.0 enum type. Change default for jaxb:globalBinding @typeEnumBase from xs:NCName to xs:string.
  - `ObjectFactory` factory methods no longer throws `JAXBException`.
  - Added customizations
    [jaxb:globalBindings] @generateValueClass, @generateElementClass, @serializable, @optionalProperty, @nullInCollection
    [jaxb:property] @generateElementProperty
  - Add binding support for redefine
  - Simplified following bindings:
    - union by binding to String rather than Object.
    - Attribute Wildcard binds to portable abstraction of a java.util.Map<QName, String>, not javax.xml.bind.AttributeMap.
    - bind xsd:anyType to java.lang.Object in JAXB property method signatures and element factory method(support element/type substitution)
  - Changes required for default and customized binding in order to support flexible unmarshalling described in Section 4.4.3.
- Java to XSD Binding
  - Added @XmlAccessorType for controlling whether fields or properties are mapped by default.
  - Added @XmlEnum and @XmlEnumValue for mapping of enum types.
  - Collections has been redesigned to allow them to be used in annotation of schema derived code:

    - removed @XmlCollectionItem and @XmlCollection

    - Added annotations parameters to @XmlElement

    - added @XmlElementRef

    - added @XmlElements and @XmlElementRefs as containers for collections of @XmlElements or @XmlElementRefs.

- added @XmlElementWrapper for wrapping of collections.
- Added mapping of anonymous types.
- Added mapping of nested classes to schema
- Added @XmlRootElement for annotating classes. @XmlElement can now only be used to annotate properties/fields.
- Added @XmlElementRef for supporting schema derived code as well as mapping of existing object model to XML representation. javadoc for @XmlElementRef contains an example
- Added @XmlElementDecl on object factory methods for supporting mapping of substitution groups for schema -> java binding.
- Redesigned Adapter support for mapping of non Java Beans.

  - new package javax.xml.bind.annotation.adapters for adapters.

  - Added XmlAdapter base abstract class for all adapters.

  - redesigned and moved XmlJavaTypeAdapter to the package.
- Moved default mapping from each section to "Default Mapping" section.
- Consistent treatment of defaults "##default"

- Removed JAX-RPC 1.1 Alignment. JAX-WS 2.0 is deferring its databinding to JAXB 2.0.

I.5 Changes for 2.0 Early Draft v0.4

- Updated Chapter 1, "Introduction".
- Added Chapter 2, "Requirements"
- Added Chapter 8, "Java Types To XML" for Java Source to XML Schema mapping.
- XML Schema to schema-derived Java Binding changes
  - Element handling changes to support element and type substitution in Section 5.6.3, "Java Element Representation Summary", Section 6.7, "Element Declaration" and Section 5.5.5, "Element Property".
  - Added Section 6.9, "Attribute Wildcard" binding
  - Support binding all wildcard content in Section 6.12.5, "Bind wildcard schema component".

  - Addition/changes in Table 6-1, "Java Mapping for XML Schema Built-in Types".
- XML Schema to Java Customization
  - Added ability to doable databinding for an XML Schema fragment in Section 7.12, "<dom> Declaration".

I.6 Changes for 1.0 Final

- Added method `javax.xml.bind.Marshaller.getNode(Object)` which returns a DOM view of the Java content tree. See method's javadoc for details.

I.7 Changes for Proposed Final

- Added Chapter 9, "Compatibility."
- Section 5.9.2, "General Content Property," removed value content list since it would not be tractable to support when type and group substitution are supported by JAXB technology.
- Added the ability to associate implementation specific property/value pairs to the unmarshal, validation and JAXB instance creation. Changes impact Section 3.4 "Unmarshalling," Section 3.5 "Validator" and the ObjectFactory description in Section 4.2 "Java Package."
- Section 6.12.10.1, "Bind a Top Level Choice Model Group" was updated to handle Collection properties occurring within a Choice value class.
- Section 6.12.11, "Model Group binding algorithm" changed step 4(a) to bind to choice value class rather than choice content property.
- Section 5.5.2.2, "List Property and Section 5.5.4, "`isSet` Property Modifier" updated so one can discard set value for a List property via calling unset method.
- At end of Section 4, added an UML diagram of the JAXB Java representation of XML content.

- Updated default binding handling in Section 6.5, "Model Group Definition." Specifically, value class, element classes and enum types are derived from the content model of a model group definition are only bound once, not once per time the group is referenced.
- Change Section 6.12.5, "Bind wildcard schema component," to bind to a JAXB property with a basetype of `java.lang.Object`, not `javax.xml.bind.Element`. Strict and lax wildcard validation processing allows for contents constrained only by `xsi:type` attribute. Current APIs should allow for future support of `xsi:type`.
- Simplify anonymous simple type definition binding to typesafe enum class. Replace incomplete approach to derive a name with the requirement that the @name attribute for element typesafeEnumClass is mandatory when associated with an anonymous simple type definition.
- Changed Section 6.5.3, "Deriving Class Names for Named Model Group Descendants" to state that all classes and interfaces generated for XML Schema component that directly compose the content model for a model group, that these classes/interfaces should be generated once as top-level interface/class in a package, not in every content model that references the model group.
- Current Section 7.5, "`<globalBindings>` Declaration":
  - Replaced `modelGroupAsClass` with `bindingStyle`.
  - Specified schema types that cannot be listed in `typesafeEnumBase`.
- Section 7.8, "`<property>` Declaration:
  - Clarified the customization of model groups with respect to `choiceContentProperty, elementBinding and modelGroupBinding`. Dropped `choiceContentProperty` from the `<property>` declaration.
  - Added `<baseType>` element and clarified semantics.
  - Added support for customization of simple content.
  - Added customization of simple types at point of reference.
  - Clarified restrictions and relationships between different customizations.
- Section 7.9, "`<javaType>` Declaration":
  - Added `javax.xml.bind.DatatypeConverterInterface`

interface.

- Added `javax.xml.bind.DatatypeConverter` class for use by user specified parse and print methods.
- Added `javax.xml.namespace.NamespaceContext` class for processing of QNames.
- Clarified print and parse method requirements.
- Added narrowing and widening conversion requirements.

- Throughout Chapter 7, "Customizing XML Schema to Java Representation Binding," clarified the handling of invalid customizations.

I.8 Changes for Public Draft 2

Many changes were prompted by inconsistencies detected within the specification by the reference implementation effort. Change bars indicate what has changed since Public Draft.

- Section 4.5.4, "isSetProperty Modifier," describes the customization required to enable its methods to he generated.
- Section 5.7.2, "Binding of an anonymous type definition," clarifies the generation of value class and typesafe enum classes from an anonymous type definition.
- Section 5.2.4, "List" Simple Type Definition and the handling of list members within a union were added since public draft.
- Clarification on typesafe enum global customization "generateName" in Section 5.2.3.4, "XML Enumvalue To Java Identifier Mapping."
- Clarification of handling binding of wildcard content in Section 5.9.4.
- Chapter6, "Customization," resolved binding declaration naming inconsistencies between specification and normative binding schema.
- removed `enableValidation` attribute (a duplicate of `enableFailFastCheck`) from `<globalBindings>` declaration.
- Added default values for `<globalBindings>` declaration attributes.

- Changed `typesafeEnumBase` to a list of QNames. Clarified the binding to typesafe enum class.
- Clarified the usage and support for `implClass` attribute in `<class>` declaration.
- Clarified the usage and support for `enableFailFastCheck` in the `<property>` declaration.
- Added `<javadoc>` to typesafe enum class, member and property declarations.
- Mention that embedded HTML tags in `<javadoc>` declaration must be escaped.
- Fixed mistakes in derived Java code throughout document.
- Added Section 7. Compatibility and updated Appendix E.2 “Non required XML Schema Concepts” accordingly.

I.9 Changes for Public Draft

- Section 6.12.7.2, “Bind single occurrence choice group to a choice content property,” replaced overloading of choice content property setter method with a single setter method with a value parameter with the common type of all members of the choice. Since the resolution of overloaded method invocation is performed using compile-time typing, not runtime typing, this overloading was problematic. Same change was made to binding of union types.
- Added details on how to construct factory method signature for nested content and element classes.
- Section 3.3, default validation handler does not fail on first warning, only on first error or fatal error.
- Add ID/IDREF handling in section 5.
- Updated name mapping in appendix C.
- section 5.5.2.1 on page 58, added getIDLenth() to indexed property.
- Removed ObjectFactory.setImplementation method from Section 5.2, “Java Package,” on page 50. The negative impact on implementation provided to be greater than the benefit it provided the user.

- Introduced external binding declaration format.
- Introduced a method to introduce extension binding declarations.
- Added an appendix section describing JAXB custom bindings that align JAXB binding with JAX-RPC binding from XML to Java representation.
- Generate isID() accessor for boolean property.
- Section 6, Customization has been substantially rewritten.